didery.py Documentation

Release 0.0.1

Nicholas Telfer

May 10, 2019

Table of Contents

This project is meant to be used in tandem with didery servers. It provides a python library and cli for communicating with didery servers.

CHAPTER 1

CLI

Command line interface that utilizes the didery.py library to communicate with didery servers

1.1 Getting Started

You will need python 3.6 and libsodium installed to run didery.py. You can find python 3.6 here and libsodium here. It is recommended that you also setup a python virtual environment as shown here.

1.1.1 Installation

To install didery.py start your virtual environment and run the command below:

```
$ pip install -e didery.py/
```

1.1.2 Usage

To see the command line options use the command below:

```
$ didery --help
```

```
Usage: didery [OPTIONS] CONFIG

Options:
  -i, --incept      Send a key rotation history inception event.
  -u, --upload      Upload a new otp encrypted private key.
  -r, --rotate      Rotate public/private key pairs.
  -U, --update      Update otp encrypted private key.
  -R, --retrieve    Retrieve key rotation history.
  -d, --download    Download otp encrypted private key.
  -D, --delete      Delete rotation history.
```

(continues on next page)

(continued from previous page)

```
  -m, --remove      Remove otp encrypted private key.
  -e, --events      Pull a record of all history rotation events for a
                    specified did.
  -v                Verbosity of console output. There are 5 verbosity levels
                    from '' to '-vvvv.'
  -M, --mute        Mute all console output except prompts.
  --data PATH       Path to the data file.
  --did TEXT        decentralized identifier(did).
  --save DIRECTORY  Directory to store generated key files in.
  --help            Show this message and exit.
```

Config File

The CLI requires a path to a json formatted config file with a list of didery endpoints as shown below.

```
{
    "servers": ["http://localhost:8080", "http://localhost:8000"]
}
```

"servers" [list] *required* - A list of server address strings. This must be supplied so the library knows what servers to broadcast and poll from. To determine if there is a consensus on polling a 2/3 of the servers must return matching responses.

Data File

For certain commands it is necessary to supply a data file. The file should be json formatted and will contain either the rotation history or the one time pad(otp) encrypted blob. The data file is required for the following options:

–upload

–rotate

The file should follow the format below for history data:

```
{
    "history": {
        "id": "did:dad:Qt27fThWoNZsa88VrTkep6H-4HA8tr54sHON1vWl6FE=",
        "signer": 0,
        "signers":
        [
            "Qt27fThWoNZsa88VrTkep6H-4HA8tr54sHON1vWl6FE=",
            "Xq5YqaL6L48pf0fu7IUhL0JRaU2_RxFP0AL43wYn148="
        ]
    }
}
```

"id" [string] *required* - Decentralized identifier (DID).

"signer" [integer] *required* - 0 based index into signers field. Rotation events signer field will always be 1 or greater.

"signers" [list] *required* - List of all public keys. Must contain at least two keys for –upload and 3 or more for –rotation.

The file should follow the format below for otp data:

```
{
    "otp": {
        "blob":
↪"AeYbsHot0pmdWAcgTo5sD8iAuSQAfnH5U6wiIGpVNJQQoYKBYrPPxAoIc1i5SHCIDS8KFFgf8i0tDq8XGizaCgo9yjuKHHNJZI
↪4D-7s3CcYmuoWAh6NVtYaf_GWw_2sCrHBAA2mAEsml3thLmu50Dw",
        "id": "did:dad:Qt27fThWoNZsa88VrTkep6H-4HA8tr54sHON1vWl6FE="
    }
}
```

"id" [string] *required*
- Decentralized identifier (DID).

"blob" [string] *required*
- otp encrypted private keys.

1.2 Example Config File

1.2.1 config.json

```
{
    "servers": ["http://localhost:8080", "http://localhost:8000"]
}
```

1.3 Example Data File

A data file can contain a history json object, a otp json object, or both. You can specify the path to the file using the –data cli option.

1.3.1 data.json

```
{
    "history": {
        "id": "did:dad:LYyYqfpFLbRcqqah3ViCBPl-c0wW5qo7IpT9Fl13I4Q=",
        "signer": 1,
        "signers":
        [
            "LYyYqfpFLbRcqqah3ViCBPl-c0wW5qo7IpT9Fl13I4Q=",
            "CQPaPAhXN0zS0pP94ms1usKlCPUK1GBXBlCSlXMX02U=",
            "qofdqNFvYbi52ZzaVM9hB0i8hUNbUQRZkhpHFpyYcfU="
        ]
    },
    "otp": {
        "blob":
↪"AeYbsHot0pmdWAcgTo5sD8iAuSQAfnH5U6wiIGpVNJQQoYKBYrPPxAoIc1i5SHCIDS8KFFgf8i0tDq8XGizaCgo9yjuKHHNJZI
↪4D-7s3CcYmuoWAh6NVtYaf_GWw_2sCrHBAA2mAEsml3thLmu50Dw",
        "id": "did:dad:LYyYqfpFLbRcqqah3ViCBPl-c0wW5qo7IpT9Fl13I4Q="
```

(continues on next page)

(continued from previous page)

```
        }
}
```

CHAPTER 2

Library

Python library for generating keys and broadcasting or polling didery servers.

2.1 Getting Started

You will need python 3.6 and libsodium installed to run didery.py. You can find python 3.6 here and libsodium here.

2.1.1 Installation

To install didery.py start your virtual environment and run the command below:

```
$ pip install -e didery.py/
```

2.1.2 Importing

```
import diderypy.lib as lib

vk, sk, = lib.generating.keyGen()

print(vk)
print(sk)
```

2.2 didering.py

This module provides various DID generation and manipulation functions for use with the didery server.

2.2.1 didering.didGen(vk, [method])

didGen accepts an EdDSA (Ed25519) key in the form of a byte string and returns a DID.

vk (*required*)- 32 byte verifier/public key from EdDSA (Ed25519) key
method (*optional*) - W3C did method string. Defaults to "dad".

returns - W3C DID string

Example

```
import diderypy.lib.didering as did


vk = b'\xfdv\xae\xeb\xe7\x08Q\xaf\xedY\xcf\x8b"\xfc\xa6\xeb\x1c@\x89}
↪\xdb\xed\x16\xa5\xb6\x88\x18\xc8\x1a%O\x83'

# use the default method
did1 = did.didGen(vk)

# or you can specify a method like igo
did2 = did.didGen(vk, "igo")

print(did1)
print(did2)
```

Output

```
did:dad:_Xau6-cIUa_tWc-LIvym6xxAiX3b7RaltogYyBolT4M=
did:igo:_Xau6-cIUa_tWc-LIvym6xxAiX3b7RaltogYyBolT4M=
```

2.2.2 didering.didGen64(vk64u, [method]):

didGen accepts a url-file safe base64 key in the form of a string and returns a DID.

vk64u (*required*)- base64 url-file safe verifier/public key from EdDSA (Ed25519) key
method (*optional*) - W3C did method string. Defaults to "dad"

returns - W3C DID string

Example

```
import diderypy.lib.didering as did


vk = "nxESHveBmK9RsEkgaZi-cNPvW0zO-ujOWEW7oKb7EYI="
```

(continues on next page)

(continued from previous page)

```
# use the default method
did1 = did.didGen64(vk)

# or you can specify a method like igo
did2 = did.didGen64(vk, "igo")

print(did1)
print(did2)
```

Output

```
did:dad:nxESHveBmK9RsEkgaZi-cNPvW0zO-ujOWEW7oKb7EYI=
did:igo:nxESHveBmK9RsEkgaZi-cNPvW0zO-ujOWEW7oKb7EYI=
```

2.2.3 didering.extractDidParts(did):

extractDidParts parses and returns a tuple containing the prefix method and key string contained in the supplied W3C DID string. If the supplied string does not fit the pattern pre:method:keystr a ValueError is raised.

did (*required*)- W3C DID string

returns - (pre, method, key string) a tuple containing the did parts.

Example

```
import diderypy.lib.didering as did


did1 = "did:dad:nxESHveBmK9RsEkgaZi-cNPvW0zO-ujOWEW7oKb7EYI="
did2 = "did:igo:nxESHveBmK9RsEkgaZi-cNPvW0zO-ujOWEW7oKb7EYI="


result1 = did.extractDidParts(did1)
result2 = did.extractDidParts(did2)

print(result1)
print(result2)
```

Output

```
('did', 'dad', 'nxESHveBmK9RsEkgaZi-cNPvW0zO-ujOWEW7oKb7EYI=')
('did', 'igo', 'nxESHveBmK9RsEkgaZi-cNPvW0zO-ujOWEW7oKb7EYI=')
```

2.2.4 didering.validateDid(did, [method]):

validateDid accepts a W3C DID string and an optional method argument. It returns the DID as well as the public/verifier key contained in the did. If the DID is invalid a ValueError is raised.

did (*required*)- W3C DID string
method (*optional*) - W3C did method string. Defaults to "dad"

returns - Tuple with W3C DID string, and the did's verifier/public key

Example

```
import diderypy.lib.didering as did


did1 = "did:dad:nxESHveBmK9RsEkgaZi-cNPvW0zO-ujOWEW7oKb7EYI="
did2 = "did:igo:nxESHveBmK9RsEkgaZi-cNPvW0zO-ujOWEW7oKb7EYI="

# use the default method
result1 = did.validateDid(did1)

# or you can specify a method like igo
result2 = did.validateDid(did2, "igo")

print(result1)
print(result2)
```

Output

```
('did:dad:nxESHveBmK9RsEkgaZi-cNPvW0zO-ujOWEW7oKb7EYI=', 'nxESHveBmK9RsEkgaZi-
↪cNPvW0zO-ujOWEW7oKb7EYI=')
('did:igo:nxESHveBmK9RsEkgaZi-cNPvW0zO-ujOWEW7oKb7EYI=', 'nxESHveBmK9RsEkgaZi-
↪cNPvW0zO-ujOWEW7oKb7EYI=')
```

2.3 generating.py

This module provides various key generation and manipulation functions for use with the didery server. Keys are generated using the python libnacl library.

2.3.1 generating.keyToKey64u(key):

keyToKey64u allows you to convert a key from a byte string to a base64 url-file safe string.

key (*required*)- 32 byte string

returns - base64 url-file safe string

Example

```
import diderypy.lib.generating as gen


vk = b'\xfdv\xae\xeb\xe7\x08Q\xaf\xedY\xcf\x8b"\xfc\xa6\xeb\x1c@\x89}
↪\xdb\xed\x16\xa5\xb6\x88\x18\xc8\x1a%O\x83'
```

(continues on next page)

(continued from previous page)

```
# convert the key
key = gen.keyToKey64u(vk)

print(key)
```

Output

```
_Xau6-cIUa_tWc-LIvym6xxAiX3b7RaltogYyBolT4M=
```

2.3.2 generating.key64uToKey(key64u):

key64uToKey allows you to convert a base64 url-file safe key string to a byte string

key64u (*required*)- base64 ulr-file safe string

returns - byte string

Example

```
import diderypy.lib.generating as gen


key64u = "nxESHveBmK9RsEkgaZi-cNPvW0zO-ujOWEW7oKb7EYI="

# convert the key
key = gen.key64uToKey(key64u)

print(key)
```

Output

```
b'\x9f\x11\x12\x1e\xf7\x81\x98\xafQ\xb0I␣
↪i\x98\xbep\xd3\xef[L\xce\xfa\xe8\xceXE\xbb\xa0\xa6\xfb\x11\x82'
```

2.3.3 generating.keyGen(seed=None):

keyGen generates a url-file safe base64 public private key pair. If a seed is not provided libnacl's randombytes() function will be used to generate a seed.

seed (*optional*)- The seed value used during key generation.

returns - url-file safe base64 verifier/public key, signing/private key

Example

```
import libnacl
import diderypy.lib.generating as gen


seed = libnacl.randombytes(libnacl.crypto_sign_SEEDBYTES)

# generate key pair with custom seed
vk, sk, did = gen.keyGen(seed)
print(vk)
print(sk)
print(did)

# generate key pair with built in seed
vk, sk, did = gen.keyGen()
print(vk)
print(sk)
print(did)
```

Output

```
0RvCaAvHInLezCP97jaHoPokAGfP5LTpwAvcR4YqNxQ=
qNrFUd0pqLbTLIIo_xXpQFuKrqFJe45GO_dMt_OqPITRG8JoC8cict7MI_3uNoeg-iQAZ8_
↪ktOnAC9xHhio3FA==
did:dad:0RvCaAvHInLezCP97jaHoPokAGfP5LTpwAvcR4YqNxQ=

0hZpSyBosXHj52TkceVdJoPGmGt26D5ErAEO0I5m-bg=
qNjuiN_
↪MijfK8eIvJJ4mf7IRMh7noEK92KAUNXzNPPXSFmlLIGixcePnZORx5V0mg8aYa3boPkSsAQ7Qjmb5uA==
did:dad:0hZpSyBosXHj52TkceVdJoPGmGt26D5ErAEO0I5m-bg=
```

2.3.4 generating.historyGen(seed=None):

historyGen generates a new key history dictionary and returns the history along with all generated keys. If a seed is not provided libnacl's randombytes() function will be used to generate a seed.

seed (*optional*)- The seed value used during key generation.

returns - - a history dictionary with an "id", "signer" and "signers" field - url-file safe base64 verifier/public key string - url-file safe base64 signing/private key - url-file safe base64 pre-rotated verifier/public key - url-file safe base64 pre-rotated signing/private key

Example

```
import libnacl
import diderypy.lib.generating as gen

seed = libnacl.randombytes(libnacl.crypto_sign_SEEDBYTES)

# generate key pair with custom seed
history, vk, sk, pvk, psk = gen.historyGen(seed)
print("History: {}".format(history))
print("public/verification key: \n{}".format(vk))
print("private/signing key: \n{}".format(sk))
```

(continues on next page)

(continued from previous page)

```
print("pre-rotated public/verification key: \n{}".format(pvk))
print("pre-rotated private/signing key: \n{}".format(psk))

# generate key pair with built in seed
history, vk, sk, pvk, psk = gen.historyGen()
print("History: \n{}".format(history))
print("public/verification key: \n{}".format(vk))
print("private/signing key: \n{}".format(sk))
print("pre-rotated public/verification key: \n{}".format(pvk))
print("pre-rotated private/signing key: \n{}".format(psk))
```

Output

```
History: {
    'id': 'did:dad:i2ZGgZbsjw0SsZPJLis5sBjBl_FBO9cAk7tOdcCtMt0=',
    'signer': 0,
    'signers': [
        'i2ZGgZbsjw0SsZPJLis5sBjBl_FBO9cAk7tOdcCtMt0=',
        'i2ZGgZbsjw0SsZPJLis5sBjBl_FBO9cAk7tOdcCtMt0='
    ]
}

public/verification key:
i2ZGgZbsjw0SsZPJLis5sBjBl_FBO9cAk7tOdcCtMt0=

private/signing key:
SiMxYSaGTF2XHx648dqNAIfSOoRfQd-
↪3SbE0sT7WE72LZkaBluyPDRKxk8kuKzmwGMGX8UE71wCTu051wK0y3Q==

pre-rotated public/verification key:
i2ZGgZbsjw0SsZPJLis5sBjBl_FBO9cAk7tOdcCtMt0=

pre-rotated private/signing key:
SiMxYSaGTF2XHx648dqNAIfSOoRfQd-
↪3SbE0sT7WE72LZkaBluyPDRKxk8kuKzmwGMGX8UE71wCTu051wK0y3Q==



History: {
    'id': 'did:dad:ognfYHtL5HLAQUox5jODI2L5R8O3coGsN3ZKEfrKRqc=',
    'signer': 0,
    'signers': [
        'ognfYHtL5HLAQUox5jODI2L5R8O3coGsN3ZKEfrKRqc=',
        'FuacQCdWImyzZwcMkIxKjoH1Kp_4SY6KsGWhc83fGrc='
    ]
}

public/verification key:
ognfYHtL5HLAQUox5jODI2L5R8O3coGsN3ZKEfrKRqc=

private/signing key:
0rmt38sxKXWwwMfhGzGmt5tCNcLOsW4_kYu5zULbGVeiCd9ge0vkcsBBSjHmM4MjYvlHw7dygaw3dkoR-
↪spGpw==

pre-rotated public/verification key:
```

(continues on next page)

(continued from previous page)

```
FuacQCdWImyzZwcMkIxKjoH1Kp_4SY6KsGWhc83fGrc=

pre-rotated private/signing key:
t9CMQT-u3VhAj7R-GuZ_UaScc_RGE7E-YgJxfIhMLAoW5pxAJ1YibLNnBwyQjEqOgfUqn_
↪hJjoqwZaFzzd8atw==
```

2.4 historying.py

This module provides methods for asynchronously broadcasting and polling multiple didery servers for rotation histories. In the event of polling from the servers the methods will automatically check for a 2/3 majority of matching responses.

2.4.1 historying.postHistory(data, sk, urls)

postHistory accepts a didery rotation history, a signing/private key, and a list of urls and returns a dictionary of url, response key pairs

data (*required*)- rotation history as specified in the didery documentation
sk (*required*)- current signing key. base64 url-file safe signing/private key from EdDSA (Ed25519) key pair
urls (*required*)- list of url strings to query

Example

```
import diderypy.lib.historying as hist
import diderypy.lib.generating as gen

# generate the rotation history
history, vk, sk, pvk, psk = gen.historyGen()

urls = ["http://localhost:8080", "http://localhost:8000"]

result = hist.postHistory(history, sk, urls)

print(result)
```

Output

```
{
    "http://localhost:8000": {
        "data": {
            "history": {
                "id": "did:dad:cF8UIyTkUYg-I0kW5VmOsvy69Usmwy4-VgNxaeM95W8=",
                "signer": 0,
                "signers": [
                    "cF8UIyTkUYg-I0kW5VmOsvy69Usmwy4-VgNxaeM95W8=",
                    "sPCgHd2yrudecNchcXXCHVybFr9HfXPIcTP0xddJBNY="
                ],
```

(continues on next page)

(continued from previous page)

```
                "changed": "2018-07-16T19:52:39.115677+00:00"
            },
            "signatures": {
                "signer": "7J2kDoAd975cDwdczE6H-
↪9HBqVPHl4mvQepsO1nhe1eH9rLZsHzv7Bd9uufmWGKEKbowMQROONSIiROMam7CDQ=="
            }
        },
        "http_status": 201
    },
    "http://localhost:8080": {
        "data": {
            "history": {
                "id": "did:dad:cF8UIyTkUYg-I0kW5VmOsvy69Usmwy4-VgNxaeM95W8=",
                "signer": 0,
                "signers": [
                    "cF8UIyTkUYg-I0kW5VmOsvy69Usmwy4-VgNxaeM95W8=",
                    "sPCgHd2yrudecNchcXXCHVybFr9HfXPIcTP0xddJBNY="
                ],
                "changed": "2018-07-16T19:52:39.115677+00:00"
            },
            "signatures": {
                "signer": "7J2kDoAd975cDwdczE6H-
↪9HBqVPHl4mvQepsO1nhe1eH9rLZsHzv7Bd9uufmWGKEKbowMQROONSIiROMam7CDQ=="
            }
        },
        "http_status": 201
    }
}
```

2.4.2 historying.putHistory(data, sk, psk, urls)

putHistory sends a rotation event to the didery servers where they verify and store the event. putHistory returns a dictionary of url, response key pairs

data (*required*)- rotation history as specified in the didery documentation
sk (*required*)- current signing key. base64 url-file safe signing/private key from EdDSA (Ed25519) key pair
psk (*required*)- pre rotated signing key. base64 url-file safe signing/private key from EdDSA (Ed25519) key pair
urls (*required*)- list of url strings to query

Example

```
import diderypy.lib.historying as hist
import diderypy.lib.generating as gen

# rotation history must already exist before sending the put request
history, vk, sk, pvk, psk = gen.historyGen()

urls = ["http://localhost:8080", "http://localhost:8000"]

hist.postHistory(history, sk, urls)
```

(continues on next page)

(continued from previous page)

```
# generate the new pre rotated key
new_pvk, new_psk, unneeded = gen.keyGen()

# add public key to history
history["signers"].append(new_pvk)

# update current signer
history["signer"] = 1

# send rotation event
result = hist.putHistory(history,sk, psk, urls)

print(result)
```

Output


```
{
    "http://localhost:8000": {
        "data": {
            "history": {
                "id": "did:dad:R_B11yIRNt19ty_Lvt8OpZuA0_Mgs1he6zPXyttl4V4=",
                "signer": 1,
                "signers": [
                    "R_B11yIRNt19ty_Lvt8OpZuA0_Mgs1he6zPXyttl4V4=",
                    "Qbf97bKWC2G5KYM0BSX4aMWiLx-Exh3FUf4E7k6i_AY=",
                    "DHowCo3BOUyxXfx9LhI9koSDI7IQwiM7aV4H7AZ6I_A="
                ],
                "changed": "2018-07-16T20:18:29.527613+00:00"
            },
            "signatures": {
                "signer": "edDONPBidBWn1gQWNIRjtKeURGAKlfH5aHm-Ib_9thqJfVAlqaS4wSl8Ru_
↪nHNU04OEgO9-FtvxQq_NXxyGmBQ==",
                "rotation": "6hsvAoZmwzqZxegm6JeYpuFPTVQIL2g0NAiF-
↪tkDdhnVBnMp2I5XC4iC7FPqsCbosTcl0Ddnaj8LkVKIzgTdCA=="
            }
        },
        "http_status": 200
    },
    "http://localhost:8080": {
        "data": {
            "history": {
                "id": "did:dad:R_B11yIRNt19ty_Lvt8OpZuA0_Mgs1he6zPXyttl4V4=",
                "signer": 1,
                "signers": [
                    "R_B11yIRNt19ty_Lvt8OpZuA0_Mgs1he6zPXyttl4V4=",
                    "Qbf97bKWC2G5KYM0BSX4aMWiLx-Exh3FUf4E7k6i_AY=",
                    "DHowCo3BOUyxXfx9LhI9koSDI7IQwiM7aV4H7AZ6I_A="
                ],
                "changed": "2018-07-16T20:18:29.527613+00:00"
            },
            "signatures": {
                "signer": "edDONPBidBWn1gQWNIRjtKeURGAKlfH5aHm-Ib_9thqJfVAlqaS4wSl8Ru_
↪nHNU04OEgO9-FtvxQq_NXxyGmBQ==",
                "rotation": "6hsvAoZmwzqZxegm6JeYpuFPTVQIL2g0NAiF-
↪tkDdhnVBnMp2I5XC4iC7FPqsCbosTcl0Ddnaj8LkVKIzgTdCA=="
```


(continues on next page)

(continued from previous page)

```
            }
        },
        "http_status": 200
    }
}
```

2.4.3 historying.getHistory(did, urls)

getHistory accepts a W3C decentralized identifier(DID) string and a list of urls to poll and returns a single rotation history if 2/3 of the urls returned matching data. If less than 2/3 returned matching data None is returned.

did (*required*)- W3C decentralized identifier(DID) string
urls (*required*)- list of url strings to query

returns - (dict, dict) containing the rotation history as shown on the didery documentation and a results dict containing a short string description for each url. The results dict can be used to determine what urls failed.

Example

```
import diderypy.lib.historying as hist
import diderypy.lib.generating as gen

# generate the rotation history
history, vk, sk, pvk, psk = gen.historyGen()

urls = ["http://localhost:8080", "http://localhost:8000"]

# history must already exist to use getHistory
hist.postHistory(history, sk, urls)

did = history["id"]

data, results = hist.getHistory(did, urls)

if data is None:
    # Consensus could not be reached. Print results for each url
    for url, result in results.items():
        print("{}:\t{}".format(url, result))
else:
    print(data)
```

Output

```
{
    "history": {
        "id": "did:dad:g3Jr_qvnh4EERpl0ohu8HNz07gw4Im666Gz7KL81U5g=",
        "signer": 0,
        "signers": [
            "g3Jr_qvnh4EERpl0ohu8HNz07gw4Im666Gz7KL81U5g=",
```

(continues on next page)

(continued from previous page)

```
            "M4t0cFPqWzg6uy2OjOZwhyNQ6rrZBO4DIO51o-Ax7wo="
        ],
        "changed": "2018-07-16T21:03:41.381008+00:00"
    },
    "signatures": {
        "signer": "TnC14l6ojngaVfmRJLqePT4YC22wgKgAd7GFDlyWswshC3G46_FNcMo4rSQxm-
↪tIFgC2VWRXQt_C6wd_HO2qDQ=="
    }
}
```

2.4.4 historying.deleteHistory(did, sk, urls)

For GDPR compliance a delete method is provided. For security reasons the data cannot be deleted without signing with the current key.

did (*required*)- W3C decentralized identifier(DID) string **sk** (*required*)- current signing key. base64 url-file safe signing/private key from EdDSA (Ed25519) key pair
urls (*required*)- list of url strings to query

returns - dict containing the rotation history that was deleted.

Example

```
import diderypy.lib.historying as hist
import diderypy.lib.generating as gen

# generate the rotation history
history, vk, sk, pvk, psk = gen.historyGen()

urls = ["http://localhost:8080", "http://localhost:8000"]

# history must already exist to use getHistory
hist.postHistory(history, sk, urls)

did = history["id"]

response = hist.deleteHistory(did, sk, urls)

print(response)
```

Output

```
{
    "http://localhost:8000": {
        "data": {
            "deleted": {
                "history": {
                    "id": "did:dad:7oW7Qev4Hz6md7ldniP_EZduufdsnP5NCGdh_7JipIg=",
                    "signer": 0,
```

(continues on next page)

(continued from previous page)


```
                    "signers": [
                        "7oW7Qev4Hz6md7ldniP_EZduufdsnP5NCGdh_7JipIg=",
                        "KoFfNTrnqhCw2vdzXqFg_gUH-bdWfWSTQoaJnf5BZBg="
                    ],
                    "changed": "2018-08-21T20:43:22.359170+00:00"
                },
                "signatures": {
                    "signer": "FNV0Eiw7K79u0o7rBQFBzE8BHIf57CebdUxki-lbkYhb-
↪7JgI9wJz0OOhnwCkWxQ_gKS4vZJTtoDW06uan-ICg=="
                }
            }
        },
        "http_status": 200
    },
    "http://localhost:8080": {
        "data": {
            "deleted": {
                "history": {
                    "id": "did:dad:7oW7Qev4Hz6md7ldniP_EZduufdsnP5NCGdh_7JipIg=",
                    "signer": 0,
                    "signers": [
                        "7oW7Qev4Hz6md7ldniP_EZduufdsnP5NCGdh_7JipIg=",
                        "KoFfNTrnqhCw2vdzXqFg_gUH-bdWfWSTQoaJnf5BZBg="
                    ],
                    "changed": "2018-08-21T20:43:22.359170+00:00"
                },
                "signatures": {
                    "signer": "FNV0Eiw7K79u0o7rBQFBzE8BHIf57CebdUxki-lbkYhb-
↪7JgI9wJz0OOhnwCkWxQ_gKS4vZJTtoDW06uan-ICg=="
                }
            }
        },
        "http_status": 200
    }
}
```


2.5 history_eventing.py

This module provides methods for asynchronously polling multiple didery servers for rotation history events. The methods will automatically check for a 2/3 majority of matching responses from didery servers.

2.5.1 history_eventing.getHistoryEvents(did, urls)

getHistoryEvents accepts a W3C decentralized identifier(DID) string and a list of urls to poll and returns all events for a rotation history. This includes the inception event and all subsequent rotations events with their corresponding signatures so you can verify that the data and the current key are all valid. All data returned from the didery servers is put through a consensus algorithm that requires a 2/3 majority of data to match. If 2/3 of the urls returned matching data a single copy of the data is returned. If a majority consensus cannot be found then None is returned. The http request results are returned as a dict of key(url) value(status) pairs.

did (*required*)- W3C decentralized identifier(DID) string

urls (*required*)- list of url strings to query

returns - (dict, dict) containing the events as shown in the output section below and a results dict containing a short string description for each url. The results dict can be used to determine what urls failed.

Example

```
import diderypy.lib.history_eventing as events
import diderypy.lib.historying as hist
import diderypy.lib.generating as gen

# rotation history must already exist before sending the put request
history, vk, sk, pvk, psk = gen.historyGen()
did = history["id"]

urls = ["http://localhost:8080", "http://localhost:8000"]

hist.postHistory(history, sk, urls)

# generate the new pre rotated key
new_pvk, new_psk, unneeded = gen.keyGen()

# add public key to history
history["signers"].append(new_pvk)

# update current signer
history["signer"] = 1

# send rotation event
hist.putHistory(history,sk, psk, urls)

data, results = events.getHistoryEvents(did, urls)

if data is None:
    # Consensus could not be reached. Print results for each url
    for url, result in results.items():
        print("{}:\t{}".format(url, result))
else:
    print(data)
```

Output

```
{
    "events": {
        "1": {
            "history": {
                "id": "did:dad:l8jrnoFp-D1SUYZtrp-McD_L2lVmBdKI1LS3hJ6D0Fc=",
                "signer": 1,
                "signers": [
                    "l8jrnoFp-D1SUYZtrp-McD_L2lVmBdKI1LS3hJ6D0Fc=",
                    "HOTSwhtdXXPBYiqtzVz2yGUzipFPjuAuEALbe0FFwzc=",
                    "KSAHDoapdn1SW2WVbqlRac3UqJp7tgMRPdjtUEx8Drw="
                ],
```

(continues on next page)

(continued from previous page)

```
                "changed": "2018-09-04T22:39:32.512473+00:00"
            },
            "signatures": {
                "signer": "9msgtbfjmCyaOkZgeW-q_N6bGUZGTZ-6z54fAf-juzhXgI0G8QfBk9P_
↪Mzr832AdXjLus1QvOjNj-It_fnsVAw==",
                "rotation": "x71A29AXGGDiDxSrPEBO4-hwQg2ILEk0XVvJyUM1OdSW15agBjmFCch3_
↪L8WtmtIUZGDzYRD3JZpXztISmF0CQ=="
            }
        },
        "0": {
            "history": {
                "id": "did:dad:l8jrnoFp-D1SUYZtrp-McD_L2lVmBdKI1LS3hJ6D0Fc=",
                "signer": 0,
                "signers": [
                    "l8jrnoFp-D1SUYZtrp-McD_L2lVmBdKI1LS3hJ6D0Fc=",
                    "HOTSwhtdXXPBYiqtzVz2yGUzipFPjuAuEALbe0FFwzc="
                ],
                "changed": "2018-09-04T22:39:32.483239+00:00"
            },
            "signatures": {
                "signer": "X76g8FU1nxTiJZFpbrLIpGFPMIcpQnQ4dwB7G_
↪AR3ksb1BCVMajzCoe2J4fXfNolOvU7i8kW7m_p6X1ETtWtCQ=="
            }
        }
    }
}
```

2.6 otping.py

This module provides methods for asynchronously broadcasting and polling multiple didery servers for one time pad(otp) encrypted blobs. In the event of polling from the servers the methods will automatically check for a 2/3 majority of matching responses.

2.6.1 otping.postOtpBlob(data, sk, urls)

postOtpBlob accepts otp blob dict, a signing/private key, and a list of urls and returns a dictionary of url, response key pairs

data (*required*)- otp encrypted blob data as specified in the didery documentation
sk (*required*)- signing key associated with the public key in the accompanying did. base64 url-file safe signing/private key from EdDSA (Ed25519) key pair
urls (*required*)- list of url strings to query

Example

```
import diderypy.lib.otping as otp
import diderypy.lib.generating as gen

# generate a did for the data
```

(continues on next page)

(continued from previous page)

```
vk, sk, did = gen.keyGen()

data = {
    "id": did,
    "blob":
↪"AeYbsHot0pmdWAcgTo5sD8iAuSQAfnH5U6wiIGpVNJQQoYKBYrPPxAoIc1i5SHCIDS8KFFgf8i0tDq8XGizaCgo9yjuKHHNJZP
↪4D-7s3CcYmuoWAh6NVtYaf_GWw_2sCrHBAA2mAEsml3thLmu50Dw"
}

urls = ["http://localhost:8080", "http://localhost:8000"]

result = otp.postOtpBlob(data, sk, urls)

print(result)
```

Output

```
{
    "http://localhost:8000": {
        "data": {
            "otp_data": {
                "id": "did:dad:V7A6qo1D8VG7ZXF2h1vVeANPHrcmljPgpBNb2c4g2wA=",
                "blob":
↪"AeYbsHot0pmdWAcgTo5sD8iAuSQAfnH5U6wiIGpVNJQQoYKBYrPPxAoIc1i5SHCIDS8KFFgf8i0tDq8XGizaCgo9yjuKHHNJZP
↪4D-7s3CcYmuoWAh6NVtYaf_GWw_2sCrHBAA2mAEsml3thLmu50Dw",
                "changed": "2018-07-16T21:16:50.056107+00:00"
            },
            "signatures": {
                "signer": "b1M0f78dfMWYBpDaM7sQujmGh1HWlcLjTW7BTrIyCoXBXsrOltEXa_K--
↪Sblox1BCoBpSZ8k0uvN0j88P12DAQ=="
            }
        },
        "http_status": 201
    },
    "http://localhost:8080": {
        "data": {
            "otp_data": {
                "id": "did:dad:V7A6qo1D8VG7ZXF2h1vVeANPHrcmljPgpBNb2c4g2wA=",
                "blob":
↪"AeYbsHot0pmdWAcgTo5sD8iAuSQAfnH5U6wiIGpVNJQQoYKBYrPPxAoIc1i5SHCIDS8KFFgf8i0tDq8XGizaCgo9yjuKHHNJZP
↪4D-7s3CcYmuoWAh6NVtYaf_GWw_2sCrHBAA2mAEsml3thLmu50Dw",
                "changed": "2018-07-16T21:16:50.056107+00:00"
            },
            "signatures": {
                "signer": "b1M0f78dfMWYBpDaM7sQujmGh1HWlcLjTW7BTrIyCoXBXsrOltEXa_K--
↪Sblox1BCoBpSZ8k0uvN0j88P12DAQ=="
            }
        },
        "http_status": 201
    }
}
```

2.6.2 otping.putOtpBlob(data, sk, urls)

putOtpBlob sends an updated otp encrypted blob to the didery servers. putOtpBlob returns a dictionary of url, response key pairs

data (*required*)- otp encrypted blob data as specified in the didery documentation
sk (*required*)- current signing key. base64 url-file safe signing/private key from EdDSA (Ed25519) key pair
urls (*required*)- list of url strings to query

Example

```
import diderypy.lib.otping as otp
import diderypy.lib.generating as gen

# make sure there is already data on the server for our did
vk, sk, did = gen.keyGen()

data = {
    "id": did,
    "blob":
↪"AeYbsHot0pmdWAcgTo5sD8iAuSQAfnH5U6wiIGpVNJQQoYKBYrPPxAoIc1i5SHCIDS8KFFgf8i0tDq8XGizaCgo9yjuKHHNJZI
↪4D-7s3CcYmuoWAh6NVtYaf_GWw_2sCrHBAA2mAEsml3thLmu50Dw"
}

urls = ["http://localhost:8080", "http://localhost:8000"]

otp.postOtpBlob(data, sk, urls)

# Update data on the server
data["blob"] =
↪"OtjioHot0pmdWAcgTo5sD8iAuSQAfnH5U6wiIGpVNJQQoYKBYrPPxAoIc1i5SHCIDS8KFFgf8i0tDq8XGizaCgo9yjuKHHNJZI
↪4D-7s3CcYmuoWAh6NVtYaf_GWw_2sCrHBAA2mAEsml3thLmu50Dw"

result = otp.putOtpBlob(data, sk, urls)

print(result)
```

Output

```
{
    "http://localhost:8000": {
        "data": {
            "otp_data": {
                "id": "did:dad:Hz3XqAcXUPhiGH_OH65DfBVikYyT8A27Oe6X203Ktp8=",
                "blob":
↪"OtjioHot0pmdWAcgTo5sD8iAuSQAfnH5U6wiIGpVNJQQoYKBYrPPxAoIc1i5SHCIDS8KFFgf8i0tDq8XGizaCgo9yjuKHHNJZI
↪4D-7s3CcYmuoWAh6NVtYaf_GWw_2sCrHBAA2mAEsml3thLmu50Dw",
                "changed": "2018-07-16T21:27:53.028815+00:00"
            },
            "signatures": {
                "signer": "-UgO0QssuQbhOKPJxB4JCqfWho1lwUh018C0Rxkk2ZI_
↪PDJKqPNfS9DwUNV1JbYeZMpO-RC-zhOdgWKxjr1dBg=="
```

(continues on next page)

(continued from previous page)

```
            }
        },
        "http_status": 200
    },
    "http://localhost:8080": {
        "data": {
            "otp_data": {
                "id": "did:dad:Hz3XqAcXUPhiGH_OH65DfBVikYyT8A27Oe6X203Ktp8=",
                "blob":
↪"OtjioHot0pmdWAcgTo5sD8iAuSQAfnH5U6wiIGpVNJQQoYKBYrPPxAoIc1i5SHCIDS8KFFgf8i0tDq8XGizaCgo9yjuKHHNJZI
↪4D-7s3CcYmuoWAh6NVtYaf_GWw_2sCrHBAA2mAEsml3thLmu50Dw",
                "changed": "2018-07-16T21:27:53.028815+00:00"
            },
            "signatures": {
                "signer": "-UgO0QssuQbhOKPJxB4JCqfWho1lwUh018C0Rxkk2ZI_
↪PDJKqPNfS9DwUNV1JbYeZMpO-RC-zhOdgWKxjr1dBg=="
            }
        },
        "http_status": 200
    }
}
```

2.6.3 otping.getOtpBlob(did, urls)

getOtpBlob accepts a W3C decentralized identifier(DID) string and a list of urls to poll. getOtpBlob returns a single otp blob if 2/3 of the urls returned matching data. If less than 2/3 returned matching data None is returned.

did (*required*)- W3C decentralized identifier(DID) string
urls (*required*)- list of url strings to query

returns - (dict, dict) containing the otp encrypted blob as shown on the didery documentation and a results dict containing a short string description for each url. The results dict can be used to determine what urls failed and why.

Example

```
import diderypy.lib.otping as otp
import diderypy.lib.generating as gen

# generate a did for the data
vk, sk, did = gen.keyGen()

data = {
    "id": did,
    "blob":
↪"AeYbsHot0pmdWAcgTo5sD8iAuSQAfnH5U6wiIGpVNJQQoYKBYrPPxAoIc1i5SHCIDS8KFFgf8i0tDq8XGizaCgo9yjuKHHNJZI
↪4D-7s3CcYmuoWAh6NVtYaf_GWw_2sCrHBAA2mAEsml3thLmu50Dw"
}

urls = ["http://localhost:8080", "http://localhost:8000"]
```

(continues on next page)

(continued from previous page)

```
# data must already exist for getOtpBlob to work
otp.postOtpBlob(data, sk, urls)

# retrieve the otp data
data, results = otp.getOtpBlob(did, urls)

if data is None:
    # Consensus could not be reached. Print results for each url
    for url, result in results.items():
        print("{}:\t{}".format(url, result))
else:
    print(data)
```

Output

```
{
    "otp_data": {
        "id": "did:dad:xe5I8KgW7OkeZ6x5oHtfx5NQyJWOnoFZ_djOZr0dGz0=",
        "blob":
↪"AeYbsHot0pmdWAcgTo5sD8iAuSQAfnH5U6wiIGpVNJQQoYKBYrPPxAoIc1i5SHCIDS8KFFgf8i0tDq8XGizaCgo9yjuKHHNJZI
↪4D-7s3CcYmuoWAh6NVtYaf_GWw_2sCrHBAA2mAEsml3thLmu50Dw", "changed": "2018-07-
↪16T21:38:04.899640+00:00"
    },
    "signatures": {
        "signer": "Az-qzuaOu1xelHU9quxPMZynZZAdc1BzqUchmJVIPUsFB7QdLBnHB_
↪CXNdGK6okkDaCaxXCsyk4icQBW_dqLDA=="
    }
}
```

2.6.4 historying.removeOtpBlob(did, sk, urls)

For GDPR compliance a delete method is provided. For security reasons the data cannot be deleted without signing with the signing key associated with the public key in the did.

did (*required*)- W3C decentralized identifier(DID) string **sk** (*required*)- current signing key. base64 url-file safe signing/private key from EdDSA (Ed25519) key pair
urls (*required*)- list of url strings to query

returns - dict containing the one time pad encrypted keys that were deleted.

Example

```
import diderypy.lib.otping as otp
import diderypy.lib.generating as gen

# generate a did for the data
vk, sk, did = gen.keyGen()

data = {
```

(continues on next page)

(continued from previous page)

```
    "id": did,
    "blob":
↪"AeYbsHot0pmdWAcgTo5sD8iAuSQAfnH5U6wiIGpVNJQQoYKBYrPPxAoIc1i5SHCIDS8KFFgf8i0tDq8XGizaCgo9yjuKHHNJZP
↪4D-7s3CcYmuoWAh6NVtYaf_GWw_2sCrHBAA2mAEsml3thLmu50Dw"
}

urls = ["http://localhost:8080", "http://localhost:8000"]

# data must already exist for getOtpBlob to work
otp.postOtpBlob(data, sk, urls)

# delete the otp encrypted data
response = otp.removeOtpBlob(did, sk, urls)

print(response)
```

Output


```
{
    "http://localhost:8000": {
        "data": {
            "deleted": {
                "otp_data": {
                    "id": "did:dad:pq4ovXgMGYILIfW9Vx55-ebugLWA-7Ii6qLnPUjZVFk=",
                    "blob":
↪"AeYbsHot0pmdWAcgTo5sD8iAuSQAfnH5U6wiIGpVNJQQoYKBYrPPxAoIc1i5SHCIDS8KFFgf8i0tDq8XGizaCgo9yjuKHHNJZP
↪4D-7s3CcYmuoWAh6NVtYaf_GWw_2sCrHBAA2mAEsml3thLmu50Dw",
                    "changed": "2018-08-02T21:45:30.795185+00:00"
                },
                "signatures": {
                    "signer": "9ZIRyzBh9WkVaksQoUlBRB_
↪Zrlg8kjcepjcOvPTSjj784uYVGusWiDkSq3nOyTp78v_eHEbzDEKFw6WscN6uAw=="
                }
            }
        },
        "http_status": 200
    },
    "http://localhost:8080": {
        "data": {
            "deleted": {
                "otp_data": {
                    "id": "did:dad:pq4ovXgMGYILIfW9Vx55-ebugLWA-7Ii6qLnPUjZVFk=",
                    "blob":
↪"AeYbsHot0pmdWAcgTo5sD8iAuSQAfnH5U6wiIGpVNJQQoYKBYrPPxAoIc1i5SHCIDS8KFFgf8i0tDq8XGizaCgo9yjuKHHNJZP
↪4D-7s3CcYmuoWAh6NVtYaf_GWw_2sCrHBAA2mAEsml3thLmu50Dw",
                    "changed": "2018-08-02T21:45:30.795185+00:00"
                },
                "signatures": {
                    "signer": "9ZIRyzBh9WkVaksQoUlBRB_
↪Zrlg8kjcepjcOvPTSjj784uYVGusWiDkSq3nOyTp78v_eHEbzDEKFw6WscN6uAw=="
                }
            }
        },
        "http_status": 200
    }
```


(continues on next page)

(continued from previous page)

```
}
```

CHAPTER 3

Decentralized Autonomic Data (DAD) and the three R's of Key Management

Author: Samuel M. Smith Ph.D.

Contributor: Vishal Gupta

2018/03/07

3.1 Abstract

This paper proposes a new class of data called *decentralized autonomic data* (DAD). The term *decentralized* means that the governance of the data may not reside with a single party. A related concept is that the trust in the data provenance is diffuse in nature. Central to the approach is leveraging the emerging *DID* (decentralized identifier) standard. The term *autonomic* means self-managing or self-regulating. In the context of data, we crystalize the meaning of self-managing to include cryptographic techniques for maintaining data provenance that make the data self-identifying, self-certifying, and self-securing. Implied thereby is the use of cryptographic keys and signatures to provide a root of trust for data integrity and to maintain that trust over transformation of that data, e.g. provenance. Thus key management must be a first order property of DADs. This includes key reproduction, rotation, and recovery. The pre-rotation and hybrid recovery methods presented herein are somewhat novel.

The motivating use of DAD is to provide provenance for streaming data that is generated and processed in a distributed manner with decentralized governance. Streaming data are typically measurements that are collected and aggregated to form higher level constructs. Applications include analytics and instrumentation of distributed web or internet of things (IoT) applications. Of particular interest is the use of DADs in self-sovereign reputation systems. A DAD seeks to maintain a provenance chain for data undergoing various processing stages that follows diffuse trust security principles including signed at rest and in motion.

Streaming data applications may impose significant performance demands on the processing of the associated data. Consequently one major goal is to use efficient mechanisms for providing the autonomic properties. This means finding minimally sufficient means for managing keys and cryptographic integrity.

Importantly this paper provides detailed descriptions of the minimally sufficient means for key reproduction, rotation, and recovery for DID leveraged DADS.

3.2 Overview

A decentralized autonomic data (DAD) item is associated with a decentralized identifier, (DID). This paper does not provided a detailed definition of DIDs but does describe how DIDs are used by a DAD. The DID syntax specification is a modification of standard URL syntax per RFC-3986. As such, it benefits from familiarity, which is a boon to adoption. One of the features of a DID is that it is a self certifying identifier in that a DID includes either a public key or a fingerprint of a public key from a cryptographic public/private key pair. Thereby a signature created with the private key can be verified using the public key provided by the DID. The inclusion of the public part of a cyptographic key pair in the DID give the DID other desirable properties. These include universal uniqueness and pseuodnynmity. Because a cryptographic key pair is generated from a large random number there is an infinitessimal chance that any two DIDs are the same (collision resistance). Another way to describe a DID is that it is a cryptonym, a cryptographically derived pseudonym.

Associated with a DID is a DID Document (DDO). The DDO provides meta-data about the DID that can be used to manage the DID as well as discover services affiliated with the DID. Typically the DDO is meant to be provided by some service. The DID/DDO model is not a good match for streaming data especially if a new DID/DDO pair would need to be created for each new DAD item. But a DID/DDO is a good match when used as the root or master identifier from which an identifier for the DAD is derived. This derived identifier is called a *derived DID* or *DDID*. Thus only one DID/DDO paring is required to manage a large number of DADs where each DAD may have a unique DDID. The syntax for a DDID is identical for a DID. The difference is that only one DDO with meta-data is needed for the root DID and all the DAD items carry any additional DAD-specific meta-data, thus making them self-contained (autonomic).

3.2.1 DID Syntax

A DID or DDID has the following required syntax:

did:*method*:*idstring*

The *method* is some short string that namespaces the DID and provides for unique behavior in the associated method specification. In this paper we will use the method *dad*.

The *idstring* must be universally unique. The *idstring* can have multiple colon ":" separated parts, thus allowing for namespacing. In this document the first part of the *idstring* is linked to the public member of a cryptographic key pair that is defined by the method. In this paper we will use a 44-character Base64 URL-File safe encoding as per RFC-4648, with one trailing pad byte of the 32-byte public verification key for an EdDSA (Ed25519) signing key pair. Unless otherwise specified Base64 in this document refers to the URL-File safe version of Base64. The URL-File safe version of Base64 encoding replaces plus "+" with minus "-" and slash "" with underscore "_".

As an example a DID using this format would be as follows:

```
did:dad:Xq5YqaL6L48pf0fu7IUhL0JRaU2_RxFP0AL43wYn148=
```

An example DID with namespaced idstring follows:

```
did:dad:Xq5YqaL6L48pf0fu7IUhL0JRaU2_RxFP0AL43wYn148=:blue
```

A DID may have optional parts including a path, query, or fragment. These use the same syntax as a URL, that is, the path is delimited with slashes, */*, the query with a question mark, *?*, and the fragment with a pound sign, *#*. When the path part is provided then the query applies to the resource referenced by the path and the fragment refers to an element in the document referenced by the path. An example follows:

```
did:dad:Xq5YqaL6L48pf0fu7IUhL0JRaU2_RxFP0AL43wYn148=/mom?who=me#blue
```

In contrast, when the path part is missing but either the query or fragment part is provided then the query and/or fragment parts have special meaning. A query without a path means the the query is an operation on either the DID

itself or the DID document (DDO). Likewise when a fragment is provided then the fragment is referencing an elemet of the DDO. An example of a DID without a path but with a query follows:

```
did:dad:Xq5YqaL6L48pf0fu7IUhL0JRaU2_RxFP0AL43wYn148=?who=me
```

As will be described later, a query part on a DID expression without a path part will enable the generation of *DDIDs* (derived DIDs)

3.2.2 Minimal DAD

A minimal DAD (decentralized autonomic data) item is a data item that contains a DID or DDID that helps uniquely identify that data item or affiliated data stream. In this paper JSON is used to represent serialized DAD items but other formats could be used instead. An example minimal trivial DAD is provided below. It is trivial because there is no data payload.

```
{
    "id": "did:dad:Xq5YqaL6L48pf0fu7IUhL0JRaU2_RxFP0AL43wYn148="
}
```

To ensure data integrity (i.e. that the data has not been tampered with) a signature that is verifiable as being generated by the private key associated with the public key in the *id* field value is appended to the DAD item. This signature verifies that the DAD item was created by the holder of the associated private key The DAD item is both self-identifing and self-certifying because the identifier value given by the *id* field is included in the signed data and is verifiable against the private key associated with the public key obtained from the associated DID in the *id* field. In the example below is a trivial DAD with an appended signature. The signature is separated from the JSON serialization with characters that may not appear in the JSON.

```
{
    "id": "did:dad:Xq5YqaL6L48pf0fu7IUhL0JRaU2_RxFP0AL43wYn148="
}
\r\n\r\n
u72j9aKHgz99f0K8pSkMnyqwvEr_3rpS_z2034L99sTWrMIIJGQPbVuIJ1cupo6cfIf_
↪KCB5ecVRYoFRzAPnAQ==
```

An example DAD with a payload follows:

```
{
    "id": "did:dad:Xq5YqaL6L48pf0fu7IUhL0JRaU2_RxFP0AL43wYn148=",
    "data":
    {
        "name": "John Smith",
        "nation": "USA"
    }
}
\r\n\r\n
u72j9aKHgz99f0K8pSkMnyqwvEr_3rpS_z2034L99sTWrMIIJGQPbVuIJ1cupo6cfIf_
↪KCB5ecVRYoFRzAPnAQ==
```

While the simple DADs given in the examples above are minimally self-identifying and self-certifying, they do not provide support for other self-management properties such as key management. In other words, because each DID (Decentralized Identifer) references a public signing key with its associated private key, it needs to be managed as a key not just as an identifier. The following sections will introduce the core key-management properties and the associated meta-data that a DAD needs in order to support those properties.

3.3 Key Management

The three main key management operations are:

- Reproduction
- Rotation
- Recovery

We call these the essential three R's of key management.

3.3.1 Key Reproduction

Key reproduction is all about managing the creation of new or derived keys. Each new DID requires a new public/private key pair. The private keys must be kept in a secured location. One reason to create unique public/private key pairs for each pair-wise relationship is to minimize the risk of exposure to exploits from the repeated use of a given key pair. Another reason to create unique key pairs for each interaction between parties is as a means for maintaining privacy through *pseudonymity*. This is discussed in more detail below. Minimizing the number of private keys that must be securely preserved for a given number of public keys simplifies management and reduces both expense and risk of exposure. To reiterate, there are two key-storage issues, one is storing public keys and the other is securely storing private keys. An exploit that captures a store of public keys may mean a loss of privacy because the expoiter can now correlate activity associated with those public keys. An exploit that captures a store of private keys means that the exploiter many now be able to use those private keys to take control of any associated resources. Consequently, one wants to avoid storing private keys as much as possible.

Privacy and Confidentiality

One desirable feature of a DAD is that it be privacy preserving. A simplified definition of privacy is that if two parties are participating in an exchange of data in a given context then the parties should not be linked to other interactions with other parties in other contexts. A simplified definition of confidentiality is that the content of the data exchanged is not disclosed to a third party. Confidentiality is usually obtained by encrypting the data. This paper does not specifically cover encryption but in general the mechanisms for managing encryption keys are very similar to those for managing signing keys.

An exchange can be private but not confidential, confidential but not private, both, or neither. A minimally sufficent means for preserving privacy is to use a DID as a pseudonymous identifier of each party to the exchange. A *pseudonynm* is a manufactured alias (e.g. identifier) that is under the control of its creator and that is used to identify a given interaction but is not linkable to other interactions by its owner. The ability of a third party to correlate an entity's behavior across contexts is reduced when the entity uses a unique DID for each context. Although there are more sophisticated methods for preserving privacy such as zero-knowledge proofs, the goal here is to use methods that are compatible with the performance demands of streaming data.

As mentioned above, the problem with using unique pseudonyms/cryptonyms for each exchange is that a large number of such identifiers may need to be maintained. Fortunately hierachically derived keychains provide a way to manage these cryptonyms with a reasonable level of effort.

Hierachical Deterministic Key Generation

As previously mentioned, reproduction has to do with the generation of new keys. One way to accomplish this is with a deterministic procedure for generating new public/private keys pairs where the private keys may be reproduced securely from some public information without having to be stored. A hierarchically deterministic (HD) key-generation algorithm does this by using a master or root private key and then generating new key pairs using a deterministic key-derivation algorithm. A derived key is expressed as a branch in a tree of parent/child keys. Each public key includes

the path to its location in the tree. The private key for a given public key in the tree can be securely regenerated using the root private key and the key path, also called a chain code. Only one private key, the root, needs to be stored.

The BIP-32 specification, for example, uses an indexed path representation for its HD *chain* code, such as, "0/1/2/0". The BIP-32 algorithm needs a master or root key pair and a chain code for each derived key. Then only the master key pair needs to be saved and only the master private key needs to be kept securely secret. The other private keys can be reproduced on the fly given the key generation algorithm and the chain code. An extended public key would include the chain code in its representation so that the associated private key can be derived by the holder of the master private key any time the extended public key is presented. This is the procedure for hardened keys.

The query part of the DID syntax may be used to represent an HD chain code or HD key path for an HD key that is derived from a root DID. This provides an economical way to specify derived DIDs (DDIDs) that are used to identify DADS. An example follows:

```
did:dad:Xq5YqaL6L48pf0fu7IUhL0JRaU2_RxFP0AL43wYn148=?chain=0\1\2
```

This expression above discloses the root public DID as well as the key derivation path or chain via the query part. For the sake of brevity this will be call an extended DID. The actual derived DDID is create by applying the HD algorithm such as:

```
did:dad:Qt27fThWoNZsa88VrTkep6H-4HA8tr54sHON1vWl6FE=
```

Thus a database of DDIDs could be indexed by DDID expressions with each value being the extended DID. Looking up the extended DID allows the holder to recreate on the fly the associated private key for the DDID without ever having to store the private key. This might look like the following:

```
{
    "did:dad:Qt27fThWoNZsa88VrTkep6H-4HA8tr54sHON1vWl6FE=":
↪"did:dad:Xq5YqaL6L48pf0fu7IUhL0JRaU2_RxFP0AL43wYn148=?chain=0\1\2",
   ...
}
```

Or given that the same DID method is used throughout:

```
{
    "Qt27fThWoNZsa88VrTkep6H-4HA8tr54sHON1vWl6FE=": "Xq5YqaL6L48pf0fu7IUhL0JRaU2_
↪RxFP0AL43wYn148=?chain=0\1\2",
   ...
}
```

The namespacing of the DID idstring also provides information that could be used to help formulate an HD path to generate a DDID. The following example shows two different DDIDs using the same public key and the same chain code but with a different extended idstring.

```
did:dad:Xq5YqaL6L48pf0fu7IUhL0JRaU2_RxFP0AL43wYn148=:blue?chain=0/1
did:dad:Xq5YqaL6L48pf0fu7IUhL0JRaU2_RxFP0AL43wYn148=:red?chain=0/1
```

Some refinements to this approach may be useful. One is the granularity of DDID allocation. A unique DDID could be used for each unique DAD or a unique DDID could be used for each unique destination party that is receiving a data stream. In this case each DAD would need an additional identifier to disambiguate each DAD sent to the same party. This can be provided with an additional field or by using the DID path part to provide a sequence number. This is shown in the following example:

```
did:dad:Qt27fThWoNZsa88VrTkep6H-4HA8tr54sHON1vWl6FE=/10057
```

The associated DAD is as follows:

```
{
    "id": "did:dad:Qt27fThWoNZsa88VrTkep6H-4HA8tr54sHON1vWl6FE=/10057",
    "data":
    {
        "temp": 50,
        "time": "12:15:35"
    }
}
\r\n\r\n
u72j9aKHgz99f0K8pSkMnyqwvEr_3rpS_z2034L99sTWrMIIJGQPbVuIJ1cupo6cfIf_
↪KCB5ecVRYoFRzAPnAQ==
```

Change Detection

Stale DAD items must often be detectable to prevent replay attacks. A later re-transmission of an old copy of the DAD item must not supercede a newer copy. Using a sequence number or some other identifier could provide change detection. Another way to provide change detection is for the DAD item to include a *changed* field whose value is monotonically increasing and changes every time the data is changed. The souce of the data can enforce that the *changed* field value is monotonically increasing. Typical approaches include a monotonically increasing date-time stamp or sequence number. Any older data items resent or replayed would have older date-time stamps or lower sequence numbers and would thus be detectable as stale.

Below is an example of an non-trivial data item that has a *changed* field for change detection.

```
{
    "id": "did:dad:Qt27fThWoNZsa88VrTkep6H-4HA8tr54sHON1vWl6FE=/10057",
    "changed" : "2000-01-01T00:00:00+00:00",
    "data":
    {
        "temp": 50,
        "time": "12:15:35"
    }
}
\r\n\r\n
u72j9aKHgz99f0K8pSkMnyqwvEr_3rpS_z2034L99sTWrMIIJGQPbVuIJ1cupo6cfIf_
↪KCB5ecVRYoFRzAPnAQ==
```

Change detection prevents replay attacks in the following manner. A second party receives DAD updates that are each signed by the associated private key. Each update has a monitonically increasing changed field. The source signer controls the contents of the data wrapped by the signature. Therefore the signer controls any changed field. A consistent signer will use a monotonically increasing changed value whenever the data wrapped by the signature is changed. Thus a malicious third party cannot replay earlier instances of the DAD wrapped by a valid signature to the orginal second party because the second party knows to discard any receptions that have older changed fields than the latest one they have already received.

On the Fly DDIDS in DADs

One important use case for DDIDs in DADs is to identify data that is received from a source that is not providing identifying information with the data. The receiver then creates an associated DID and DDIDs to identify the data. At some later point the receiver may be able to link this data with some other identifying information or the source may "claim" this data by supplying identifying information. In this case the DDIDs are private to the receiver but can later be used to credibly provenance the internal use of the data. This may be extremely beneficial when shared amongst the entities in the processing chain as a way to manage the entailed proliferation of keys that may all be claimed later

as a hierarchial group. The DIDs and associated derivation operations for DDIDS may be shared amongst a group of more-or-less trusted entities that are involved in the processing chain.

Public Derivation

Another important used case for DDIDS in DADS is to avoid storing even the DDID with its derivation chain. This may be an issue when a client wishes to communicate with a potenially very large number of public services. Each public service would be a new pairing with a unique DDID. If the derivation algorithm for an HD-Key DDID could use the public key or public DID of the public service to generate the DDID then the client need not store the actual DDID but can recover the DDID by using the public DID of the server to re-derive the associated DDID. This can be done by creating a hash of the root DID private key and the remote server public DID to create the seed used to generate the DDID for the DAD. This also means that the DDIDs or chain codes do not have to be included in the keys preserved by a key-recovery system.

3.3.2 Key Rotation

The simplest approach to key rotation is to revoke and replace the key in one operation. In some cases revocation without replacement is warranted. But this is the same as revoking and then replacing with a null key. Key rotation without revocation usually poses a security risk so it is not needed. Hence we simplify key management to include revocation as a subset of rotation.

Key rotation is necessary because keys used for signing (and/or encryption) may suffer increased risk of becoming compromised due to continued use over time, may be vulnerable to brute force attack merely due to advances in computing technology over time, or may become compromised due to misuse or a specific exploit. Periodically rotating the key bounds the risk of compromise resulting from exposure over time. The more difficult problem to solve is secure rotation after a specific exploit may have already occurred. In this case, the receiving party may recieve a valid signed rotation operation from the exploiter prior to the orignal holding entity sending a valid rotation operation. The receiver may erroneously accept a rotation operation that transfers control of the data to the exploiter. A subsequent rotation operation from the original holder would either create a conflict or a race condition for the receiver.

Although there are several ways to solve the early rotation exploit problem described above, the goal is to find the minimally sufficient means for preventing that exploit that is compatible with the demands of streaming data applications for which DADs are well suited.

Basic Pre-rotation

A complication with DADs is that there are two types of keys being used: the keys for the root DIDs and the keys for the derived DIDS (DDIDS). Generating a derived key pair requires using the private root key. The process for pre-rotating the root DID is described first, followed by the additional measures for DDID pre-rotation.

The approach presented here is to pre-rotate the DID key and declare the pre-rotation at the inception of the DID. This pre-rotation is declared at initialization. This may be done with an inception event. A later rotation operation event creates the next pre-rotated key thus propogating a new set of current key and pre-rotated key.

Shown below is an example inception-event data structure with a signing key in the *signer* field and a pre-rotated next signing key in the *ensuer* field. The signature is generated using the *signer* key.

Example inception event:

```
{
    "id": "did:dad:Qt27fThWoNZsa88VrTkep6H-4HA8tr54sHON1vWl6FE=",
    "changed" : "2000-01-01T00:00:00+00:00",
    "signer": "Qt27fThWoNZsa88VrTkep6H-4HA8tr54sHON1vWl6FE=",
    "ensuer": "Xq5YqaL6L48pf0fu7IUhL0JRaU2_RxFP0AL43wYn148="
```

(continues on next page)

(continued from previous page)

```
}
\r\n\r\n
u72j9aKHgz99f0K8pSkMnyqwvEr_3rpS_z2034L99sTWrMIIJGQPbVuIJ1cupo6cfIf_
↪KCB5ecVRYoFRzAPnAQ==
```

A useful convention would be that if a signer field is not provided then the signer is given by the *id* field.

```
{
    "id": "did:dad:Qt27fThWoNZsa88VrTkep6H-4HA8tr54sHON1vWl6FE=",
    "changed" : "2000-01-01T00:00:00+00:00",
    "ensuer": "Xq5YqaL6L48pf0fu7IUhL0JRaU2_RxFP0AL43wYn148="
}
\r\n\r\n
u72j9aKHgz99f0K8pSkMnyqwvEr_3rpS_z2034L99sTWrMIIJGQPbVuIJ1cupo6cfIf_
↪KCB5ecVRYoFRzAPnAQ==
```

When rotation occurs sometime later, the rotation operation atomically indicates that the key in the *signer* field is to be replaced with the pre-declared rotation key in the *ensuer* field and also declares the next rotation key to be placed in the *ensuer* field. One way to keep track of this is to provide three keys in the rotation event, the former signer in a new *erster* field, the former *ensuer* in the *signer* field and a new pre-rotated key in the *ensuer* field. The rotation operation has two signatures. The first signature is created with the former *signer* key (now *erster* field). The second signature with the former *ensuer* key (now *signer* field). This establishes provenance of the rotation operation.

Example rotation event:

```
{
    "id": "did:dad:Qt27fThWoNZsa88VrTkep6H-4HA8tr54sHON1vWl6FE=",
    "changed" : "2000-01-01T00:00:00+00:00",
    "erster": "Qt27fThWoNZsa88VrTkep6H-4HA8tr54sHON1vWl6FE=",
    "signer": "Xq5YqaL6L48pf0fu7IUhL0JRaU2_RxFP0AL43wYn148=",
    "ensuer": "dZ74MLZXD-1QHoa73w9pQ9GroAvxqFi2RTZWlkC0raY="
}
\r\n\r\n
jc3ZXMA5GuypGWFEsxrGVOBmKDtd0J34UKZyTIYUMohoMYirR8AgH5O28PSHyUB-
↪UlwfWaJlibIPUmZVPTG1DA==
\r\n\r\n
efIU4jplMtZzjgaWc85gLjJpmmay6QoFvApMuinHn67UkQZ2it17ZPebYFvmCEKcd0weWQONaTO-
↪ajwQxJe2DA==
```

Instead of three fields in the structure a list or tuple of three fields could be used where the order corresponds to `[erster, signer, ensuer]`.

In order to verify provenance over multiple rotation operations, the receiver needs to be able to replay the history of rotation operations.

The pre-rotation approach has some useful features. For many exploits, the likelihood of exploit is a function of exposure to continued monitoring or probing. Narrowly resticting the opportunity for exploits in terms of time, place, and method, especially if the time and place is a one-time event, makes exploits extremely difficult. The exploiter has to either predict the time and place of the event or has to have continuous universal monitoring of all events. By declaring the pre-rotation at the inception event of the associated DAD, the window for exploits is as narrow as possible. Pre-rotation does not require any additional keys or special purpose keys for rotation. This makes the approach self-contained. Because the rotation-operation event requires two signatures, one using the current key and the other using the pre-rotated key, an exploiter would have to exploit both keys. This is extremely difficult because the only times the private side of the pre-rotated key is used are (1) at its creation in order to make the associated public key, and (2) at the later signing of the rotation operation event. This minimizes the times and places to a narrow sample.

Listed Rotation Key Structure

Another approach to declaring rotation events is to provide the full rotation history in the rotation operation and/or to use a list structure for providing the keys. In many cases, rotations are a rare event so the number of entries in the rotation history would be small. In the associated data structure a list of all the signers both former and future to date is provided in the *signers* field. The current signer is indicated by an index into the list in the *signer* field. The list index is zero based. The pre-rotated next signer or ensuer is the following entry in the *signers* list. A rotation event then changes the signer field index, which implies that the former signer (*erster*) is the previous entry and the next pre-rotated signer (*ensuer*) is the subsequent entry after the signer index. This is shown in the following examples.

Example pre-rotated inception event with list structure for signing keys:

```
{
    "id": "did:dad:Qt27fThWoNZsa88VrTkep6H-4HA8tr54sHON1vWl6FE=",
    "changed" : "2000-01-01T00:00:00+00:00",
    "signer": 0,
    "signers":
    [
        "Qt27fThWoNZsa88VrTkep6H-4HA8tr54sHON1vWl6FE=",
        "Xq5YqaL6L48pf0fu7IUhL0JRaU2_RxFP0AL43wYn148=",
    ]
}
\r\n\r\n
jc3ZXMA5GuypGWFEsxrGVOBmKDtd0J34UKZyTIYUMohoMYirR8AgH5O28PSHyUB-
↪UlwfWaJlibIPUmZVPTG1DA==
```

The signature above is with key at index = signer = 0.

Example rotation event with list structure for signing keys:

```
{
    "id": "did:dad:Qt27fThWoNZsa88VrTkep6H-4HA8tr54sHON1vWl6FE=",
    "changed" : "2000-01-01T00:00:00+00:00",
    "signer": 1,
    "signers":
    [
        "Qt27fThWoNZsa88VrTkep6H-4HA8tr54sHON1vWl6FE=",
        "Xq5YqaL6L48pf0fu7IUhL0JRaU2_RxFP0AL43wYn148=",
        "dZ74MLZXD-1QHoa73w9pQ9GroAvxqFi2RTZWlkC0raY="
    ]
}
\r\n\r\n
jc3ZXMA5GuypGWFEsxrGVOBmKDtd0J34UKZyTIYUMohoMYirR8AgH5O28PSHyUB-
↪UlwfWaJlibIPUmZVPTG1DA==
\r\n\r\n
efIU4jplMtZzjgaWc85gLjJpmmay6QoFvApMuinHn67UkQZ2it17ZPebYFvmCEKcd0weWQONaTO-
↪ajwQxJe2DA==
```

The first signature is with key at index = signer - 1 = 0. The second signature is with key at index = signer = 1.

A subsequent rotation would add another key to the signers list and increment the signer index as follows:

```
{
    "id": "did:dad:Qt27fThWoNZsa88VrTkep6H-4HA8tr54sHON1vWl6FE=",
    "changed" : "2000-01-01T00:00:00+00:00",
    "signer": 2,
    "signers":
    [
```

(continues on next page)

(continued from previous page)

```
        "Qt27fThWoNZsa88VrTkep6H-4HA8tr54sHON1vWl6FE=",
        "Xq5YqaL6L48pf0fu7IUhL0JRaU2_RxFP0AL43wYn148=",
        "dZ74MLZXD-1QHoa73w9pQ9GroAvxqFi2RTZWlkC0raY=",
        "3syVH2woCpOvPF0SD9Z0bu_OxNe2ZgxKjTQ961LlMnA="
    ]
}
\r\n\r\n
AeYbsHot0pmdWAcgTo5sD8iAuSQAfnH5U6wiIGpVNJQQoYKBYrPPxAoIc1i5SHCIDS8KFFgf8i0tDq8XGizaCg==
\r\n\r\n
o9yjuKHHNJZFi0QD9K6Vpt6fP0XgXlj8z_4D-7s3CcYmuoWAh6NVtYaf_GWw_
↪2sCrHBAA2mAEsml3thLmu50Dw==
```

Multi-signature Pre-rotation

The list structure enables the declaration of several pre-rotations in advance by providing several future pre-rotation keys in the inception event. A rotation event then could include several rotations at once. Each rotation event would require a signature per each of the multiple rotations in the event thus allowing for multi-signature inception and rotations. If each key is from a different entity, then the rotation would require multiple entities to agree. Thus a DAD could be multi-signature and support multi-signature rotations. In this case the signer field would be a list of indices into the signers list. This approach could be further extended to support an M-of-N signature scheme where any M-of-N signatures are required to incept or rotate where M < N, and M, N are integers. The total number of keys in the list is a multiple of N. The following examples provide an inception and rotation event for a two signature pre-rotation. A namespaced key with a colon-separated idstring, as per the DID syntax, could be used to allow for signers using a different DID method or for namespacing within a given DID method.

Example of a pre-rotated two-signature inception event with list structure for signing keys where "*blue*" indicates one source and "*red*" indicates another source:

```
{
    "id": "did:dad:Qt27fThWoNZsa88VrTkep6H-4HA8tr54sHON1vWl6FE=",
    "changed" : "2000-01-01T00:00:00+00:00",
    "signer": [0,1],
    "signers":
    [
        "Qt27fThWoNZsa88VrTkep6H-4HA8tr54sHON1vWl6FE=:blue",
        "Xq5YqaL6L48pf0fu7IUhL0JRaU2_RxFP0AL43wYn148=:red",
        "dZ74MLZXD-1QHoa73w9pQ9GroAvxqFi2RTZWlkC0raY=:blue",
        "3syVH2woCpOvPF0SD9Z0bu_OxNe2ZgxKjTQ961LlMnA=:red"
    ]
}
\r\n\r\n
AeYbsHot0pmdWAcgTo5sD8iAuSQAfnH5U6wiIGpVNJQQoYKBYrPPxAoIc1i5SHCIDS8KFFgf8i0tDq8XGizaCg==
\r\n\r\n
o9yjuKHHNJZFi0QD9K6Vpt6fP0XgXlj8z_4D-7s3CcYmuoWAh6NVtYaf_GWw_
↪2sCrHBAA2mAEsml3thLmu50Dw==
```

The signatures above are generated with the keys at indices 0 and 1 in the signers list respectively.

Example of a two-signature rotation event with list structure for signing keys where "*blue*" indicates one source and "*red*" indicates another source:

```
{
    "id": "did:dad:Qt27fThWoNZsa88VrTkep6H-4HA8tr54sHON1vWl6FE=",
    "changed" : "2000-01-01T00:00:00+00:00",
    "signer": [2,3],
```

(continues on next page)

(continued from previous page)

```
    "signers":
    [
        "Qt27fThWoNZsa88VrTkep6H-4HA8tr54sHON1vWl6FE=:blue",
        "Xq5YqaL6L48pf0fu7IUhL0JRaU2_RxFP0AL43wYn148=:red",
        "dZ74MLZXD-1QHoa73w9pQ9GroAvxqFi2RTZWlkC0raY=:blue",
        "3syVH2woCpOvPF0SD9Z0bu_OxNe2ZgxKjTQ961LlMnA=:red"
        "rTkep6H-4HA8tr54sHON1vWl6FEQt27fThWoNZsa88V=:blue",
        "7IUhL0JRaU2_RxFP0AL43wYn148Xq5YqaL6L48pf0fu=:red",
    ]
}
\r\n\r\n
AeYbsHot0pmdWAcgTo5sD8iAuSQAfnH5U6wiIGpVNJQQoYKBYrPPxAoIc1i5SHCIDS8KFFgf8i0tDq8XGizaCg==
\r\n\r\n
o9yjuKHHNJZFi0QD9K6Vpt6fP0XgXlj8z_4D-7s3CcYmuoWAh6NVtYaf_GWw_
↪2sCrHBAA2mAEsml3thLmu50Dw==
\r\n\r\n
GpVNJQQoYKBYrPPxAoIc1i5SHCIDS8KFFgf8i0tDq8XGizaCgAeYbsHot0pmdWAcgTo5sD8iAuSQAfnH5U6wiI==
\r\n\r\n
8z_4D-7s3CcYmuoWAh6NVtYaf_GWw_
↪2sCrHBAA2mAEsml3thLmu50Dwo9yjuKHHNJZFi0QD9K6Vpt6fP0XgXlj==
```

The signatures above are generated with the keys at indices 0 through 3 in the signers list respectively.

Collective Signatures

This multi-signature scheme suffers from the significant increase in the length of the attached signature block. One way to ameliorate this "bloat" is to use collective multi-signatures. A collective signature has the property that its length is not a multiple of the number of signatures it holds. Typically the maximum length of a collective signature is about double the length of a non-collective signature and does not increase significantly as more signatures are added to the collective. There is a draft IETF standard for collective signatures CoSi that might be useful for multi-signature rotation. Some useful references are here project, paper, slides. Collective signatures are a type of Schnorr multi-signature or Schnorr threshold signature.

DDID Pre-rotation

The complication for DDIDs (Derived DIDs) is that each DAD stream for each pairing of sender and receiver may have a unique DDID. Rotation of the root DID also requires rotating the DDIDs. The same pre-rotation approach, however, can be used for the DDIDs. At the inception event the root key and pre-rotation root keys are created. These keys are then used to created a set of DDIDS and pre-rotated derived keys using the root and pre-rotated root keys respectively. This does not significantly change the exploit vulnerability as the inception event is still one event. Although the pre-rotated root DID key is used to create a set of pre-rotated derived keys, it does not signicantly increase its exposure. Each rotation event then involves rotating the root DID key and all the DDID keys. The important consideration is that the number of DDIDs in the set must be determined in advance in order to create all the pre-rotated derived keys at one time. This can be managed by creating extra DDIDs and pre-rotated derived keys at the inception event. Only the public half of each of the key pairs need to be stored.

In contrast, creating additional DDIDs with pre-rotated keys at a later time requires using the pre-rotated root private key. This increases the exposure of that private key to exploits and makes it less secure for pre-rotation. When the set of pre-rotated DDIDs is consumed, a rotation-operation event may be triggered, thereby rotating the existing DDIDs and then allowing additional DDIDs to be created.

Alternatively if the pre-rotated set of DDIDs is consumed then a new DDID tree may be created with a unique new pre-rotated root key. This would create a hierachy of groups of pre-rotated DDIDs and derived keys.

Moreover, when the re-establishment and re-initialization of a DAD stream is not a high-cost or high-risk endeavor then instead of pre-rotating the DDIDs, only pre-rotate the root DID and just close down the current DAD stream and re-establish with a new DDID created by the pre-rotated key as part of the rotation event.

Finally if the exposure of the root DID is insignificant compared the exposure of the DDIDs then another approach to DDID pre-rotation could be employed. This requires a trade-off between convenience and privacy. A group of receivers could all have knowledge of the root public DID key and its pre-rotated public DID key for their unique DDIDs. This means that the members of the group could leak correlation information about the group via the shared root DID. However each member of the group could still maintain security via its unique DDID. In this case the root private DID is used to derive both the inception DDID and the pre-rotated derived key of each member. The individual members could then undergo DDID key rotation but only using the root DID not its pre-rotated key. In the rare event that the root DID needs to be rotated then each of the DDID members performs a double rotation within a rotation event. The first rotation rotates to the pre-rotated key generated using the original root DID, the second rotation is to a new set of derived and pre-rotated derived keys, each generated using the new pre-rotated root key. The first derived key in the pair is the new signer key, the second is the new pre-rotated signer key. A receiver must have knowledge of the root DID and pre-rotated root key in order to verify that the second rotation is not a forgery. This approach enables the organization and managment of DDIDs in heirarchical groups where the members of each group know about their group-root DID but that group-root DID could be a DDID of a higher level group and so on. Lower level groups only know about thier group root DID, but not any sibling groups so it can't leak information about sibling or parent groups only child groups.

Replayability

The constraint on pre-rotation is that the receiving party be able to replay the rotation events to ensure that it did not miss a rotation. This replay allows the receiver to verify the provenance chain of rotations. The question then is what are minimally sufficient means for enabling this replay capability?

There are two use cases for providing this replay capability. The first case is for online one-to-one or pairwise interactions and the other case is for offline one-to-one or equivalently one-to-many or public interactions.

In the one-to-one case, there is the sender of a DAD stream and the reciever of the stream. The initiation of the stream would involve exchanging keys for pairwise communication and would also include the establishment of the DDID used for the DAD items sent. The first DAD sent would include the DDID for the DAD as will as the pre-rotated DDID. This is the inception event. The receiver then merely needs to maintain a running log of DAD items that contain rotation events. As long as reliable communications are used between the sender and receiver, then the receiver can ensure that it has observed all rotation events by keeping its log and no imposter can later send an undetectable forged inception or rotation event. If the reciever loses its history then it must re-establish its communications channel and re-initialize. Alternatively the sender could maintain a copy of the inception and rotation event history and then provide it to the receiver upon request. The receiver would cache this history for speedier lookup. An imposter attempting to send an earlier forged inception event would be unsuccessful because only the first inception event is considered valid.

In the one-to-many, public, or offline case, the rotation history is maintained by a service. While a decentralized distributed consensus blockchain ledger could provide this service it is not the minimally sufficient means of providing this capability. The minimally sufficient means is a redundant immutable event log of inception and rotation events indexed by the DDID associated with the DAD for the given DAD stream. The constraint is that a sufficient majority of the log hosts must be non-faulty at any point in time. This includes Byzantine faults. Is is also assumed that the sender communicates with the hosts using a reliable end-to-end signed protocol. The sender broadcasts the inception event to all the redundant hosts that provide copies of the log. These hosts are called Replicants. Then either the Replicants respond to the sender with a confirmation that the event is written to their log or the sender reads the log to verify. The event history is indexed by the DDID. Each Replicant timestamps and signs each entry in each event history. Each Replicant only allows one and only one inception event per event history. Attempts by imposters to forge an earlier inception event would be denied by honest Replicants. The sender can then verify that a sufficient majority of the Replicants have captured each event and have consistent event histories. Subsequent rotation events are redundantly appended to the DDID indexed log in the same way. The receiver can then broadcast a query to the Replicants and

verify via their responses that a sufficient majority of the Replicants have the same DDID indexed event log. This eanbles both offline and one-to-many event streams.

This approach is more scalable than using a distributed consensus ledger because the Replicants do not need to communicate with each other. The inter-host agreement of the members of a distributed consensus pool is usually the limiting factor in scalablity. Morever a given receiver could be completely responsible for providing the immutable log service for its own data stream with the sender. Each receiver could choose to implement a different level of reliability. Loss of the event log means that the sender and receiver have to re-initialize and re-establish the DAD stream. Alternatively the sender could be responsible for providing a set of Replicants and make the event log available to the receiver upon request.

3.3.3 Key Recovery

Key recovery is about providing a secure way of recovering a lost private key. The important consideration here is that the recovery mechanism be compatible with streaming data applications as per DADs. Keys recovery tends to be a rare occurrence so performance demands may be less constraining. Nonetheless, finding the minimally sufficient means for key recovery is still the goal. Moreover, to be secure the private key needs to be kept secret. Because cryptographic keys are long strings of numbers they are extremely hard to remember, this means that typically private keys are stored some place besides a person's memory and are therefore subject to being lost or stolen.

If it is required or at least desirable that the DAD stream not be reinitialized due to the loss of the rotation-event history then a key-recovery mechanism would also need to provide recovery of the key-rotation history. To restate, it is not enough to just recover the original root DID but every rotated root DID must be recovered as well. Given that typically rotations happen rarely, the rotation-event history should be small in size and not pose a storage-size problem for recovery. Thus key recovery for DADs needs to at least recover the original root key and any rotations.

DDIDs can be regenerated from the root DID given the HD-derivation code. In the case where the the DDID stream may not be easily reestablished but must resume given the latest rotated DDID then the HD chain code must also be preserved and recovered. If the number of DDIDs is very large then the storage requirements for chain codes may also be large relative to the storage requirement for key recovery. The DID root public key and DDID derivation chain codes do not expose the private keys. However, although disclosing the root public key and chain code for a DDID is not a security risk, it could be a privacy risk. A third party could correlate data streams from the associated DDIDs should the root public key used by multiple DDIDs be exposed. One way to address this is to encrypt the chain codes with an encryption key derived from the root signing key. The chain codes can then be stored outside of the core recovery system. The worst case exploit then is a loss of privacy should the encryption be broken but not a loss of control of the resources owned by the private key.

When the DDID for communicating with a public service is derived from the public key of a server then the client does not need to preserve and recover the HD chain code. Instead it can regenerate the DDID using a hash of the root private DID and the public DID of the server. A complication occurs when the root private key has been rotated and the server was not made aware of the rotation. The client can still recover the current root DID used by the server using a trial and error approach by going through the list of rotated root DIDs, generating the associated DDID or derived key, verifying if the server will accept it, and if not incrementing to the next rotated root. Eventually the client will discover the last rotated DDID or derived key recognized by the serve. As a result the client can recover the appropriate DDID or derived key for a given service without having to preserve anything but the history of rotated root DIDs. This approach may provide meaningful storage savings when the number of external services is large.

Cryptographic Strength

Information Theoretic Security and Perfect Security

With respect to *DAD*, key recovery deals with the recovery of the private half of signing and/or encryption keys in public/private key pairs. Given that once an adversary has the private key, security is completely broken, the cryptosystems used to backup and recover private keys needs to be as secure as is practically possible. The highest

level of crypto-graphic secruity is called information-theoretic security. A cryptosystem that has this level of security cannot be broken algorithimically even if the adversary has nearly unlimited computing power including quantum computing. It must be broken by brute force if at all. Brute force means that in order to guarantee success the adversary must search every combination of key or seed. A special case of information-theoretic security is called perfect security. Perfect security means that the cipher text provides no information about the key. There are two well-known cryptosystems that can exhibit perfect sercurity. One is *secret sharing or splitting* (see also ss). The other is a *one-time pad* (see also otp. Correct implementation of either/or a combination of these two approaches is appropriate for private-key recovery.

Sufficient Cryptographic Strength to Withstand a Brute-force Attack

For cryptosystems with perfect security, the fundamental parameter is the number of bits of entropy needed to resist any practical brute force attack. In other words, when a large random number is used as a seed/key to a cryptosystem that has perfect security, the question to be answered is how large does the random number need to be to withstand a brute force attack? In Shannon information theory the entropy of a message is measured in bits. The randomness of a number or message can measured by the number of bits of entropy in the number. A cryptographic quality random number will have as many bits of entropy as the number of bits in the number. Assuming conventional non-quantum computers, the convention wisdom is that, for systems with information theoretic or perfect security, the seed/key needs to have on the order of 128 bits (16 bytes) to practically withstand any brute force attack. For other cryptosystems that do not have perfect security the size of the seed/key may need to be much larger.

Theoretically, quantum computers, using Grover's Algorithm might be able to brute force a 2N random number with only 2N/2 trials. Thus once quantum computers exists the size of N might need to increase from 128 to 256.

An N-bit long base-2 random number has 2N different possible values. Given that with perfect security no other information is available to an attacker, the attacker may need to try every possible value before finding the correct one. Thus the number of attempts that the attacker would have to test may be as much as 2N-1. Given available computing power, one can estimate if 128 is a large enought N to make brute force attack impractical.

Let's suppose that the adversary has access to supercomputers. Current supercomputers can perform on the order of one quadrillion operations per second. Individual CPU cores can only perform about 4 billion operatons per second but a supercomputer will employ many cores in parallel. A quadrillion is approximately 250 = 1,125,899,906,842,624. Suppose somehow an adversary had a million (220 = 1,048,576) super computers to employ in parallel. The adversary could then try 250 * 220 = 270 values per second (assuming that each try only took one operation). There are about 3600 * 24 * 365 = 313,536,000 = 2log2313536000=224.91 ~= 225 seconds in a year. Thus this set of a million super computers could try 250+20+25 = 295 values per year. For a 128-bit random number this means that the adversary would need on the order of 2128-95 = 233 = 8,589,934,592 years to find the right value. This assumes that the value of breaking the cryptosystem is worth the expense of that much computing power. Consequently, a cryptosystem with perfect security and 128 bits of cryptographic strength is practically impossible to break.

Recovery Methods

Fundamentally key recovery involves shifting the burden of remembering a cryptographic key made of a long random string of numbers to some other task that is less onerous.

Physical Security

One approach to recovery is to shift the burden of recovery from remembering a private key or keys to protecting physical copies of the keys. This is called physical security. Recovery first involves creating a hard copy of the key(s) such as a printed piece of paper or a "hard" electronic wallet and then hiding the hard copy. The memory task now becomes remembering where the hard copy was hidden. The security of the approach is now based on the physical security of the hidden location (under the bed, in the safety deposit box, in a hole in the backyard). The assumption is that remembering where something is hidden is assumed to be relatively reliable. Most important is that physical

security is not vulnerable to remote attacks over the internet nor computational attacks where the attacker can employ resources and time to break a key. The attacker must have physical access and may be physically at risk. A weakness of this approach is that recovery may take time. Moreover if the person with the knowledge of the key location is incapacitated then recovery may be impossible unless the location of hard copy or another hard copy is shared with someone else, thus exposing a vulnerability. One way to address this is to use a legal mechanism such as power of attorney, a will, or another guardian who is authorized to reveal the hard copy given predefined circumstances. This can be ameliorated by using tamper-resistant envelopes and physical access logs to increase the risk of discovery. In any event physical recovery is useful as a backup to non-physical security recovery methods but may be too inconvenient as the primary form of recovery for the managers of streaming data applications. In general physical security may be a good backup for any of the other recovery methods.

Mnemonics

A mnemonic is a device or technique to aid human memory. The memory task in this case is to remember a 128-bit random number as a key or seed. This is further complicated for DAD recovery as it is not sufficient to just recover a single private key but instead requires the recovery of the whole key rotation history. One way to accomplish this is to use a 128-bit random number as a seed to a system that hides and recovers the whole rotation history. This will be discussed in more detail below. One well-known mnemonic is to use a phrase of random words from a word list. The user can create a story or imaginery visualation of a situation in which the words are all represented. An example would be the words, *blue cat house eat pudding*. Visualizing and rehearsing a fantastic situation that includes objects and actions corresponding to the words makes is much easier to remember.

The DiceWare(see also wk and pp approach consists of a word list of 7776 words that are selected at random (using dice). The user must remember the words and their order to form a phrase that can be used to generate a random number. The EFF has produced modified versions of the word list (EFF word list) that have beneficial properties. Given a total of 7776 words, then each randomly selected word is one of 7776 choices, which provides log2(7776) = 12.9 bits of entropy per word. To get a 128 bits of entroy the phrase would need to include ten words. This is pretty long for a mnemonic but not impractical as long as the user is willing to do some rehearsal. More problematic is recovering not just one key but multiple keys from a key rotation history.

Secret Sharing

Another approach is to shift the task of recovery to other parties. This can be done securely using a secret sharing or "splitting" approach. The secret information is split into what are sometimes called shards. Each shard is then shared with another party called a shard holder. Later the shards are collected and combined to reproduce the secret. The shard holders must either keep the shard secret or if they are going to store it online they need to encrypt the shard and must then remember their encryption key. As mentioned above, secret sharing may have perfect security. This means that storing encrypted copies of the shards online may still be perfectly secure as long as an adversay cannot correlate the shards as belonging to the same secret information. If correlation does occur then the security is limited to the type of encryption and might be more vulnerable to exploits.

In order to recover the secret information the user must interact with the shard holders to get them to provide their shard; that is, the recovery is multi-party interactive. The user then combines the shards to reconstitute the shared secret. This interaction may take time and may not be reliable. A useful variation on this approach is called threshold or Shamir sharing where only a subset of all the shards is needed to reconstitute the secret. For example an M of N threshold secret sharing (M < N) algorithm would share shards with N parties. Any combination of a subset of M parties can reconstitute the secret. This allows some of the parties to not be available or to lose their shard and still have successful recovery. Typically, to maintain secrecy the N parties do not know of each other.

Although the security properties of Secret sharing make it an attractive approach for key recovery, secret sharing can be complicated, especially because it requires interaction with multiple parties. The secret owner must recall who the N parties are or at least M non-faulty parties. In an organizational setting, however, there may be a designated group of individuals who know about and hold the shards and have a policy for circumstances under which they can share the shards.

One-time Pad

As mentioned previously, the *one-time pad* (OTP) (see also otp) may exhibit perfect security. The OTP is a venerable cyphersystem that has the advantage that it can be used manually without a computer. Basically a long string of random characters forms the *pad*. Someone can use the pad to encrypt a plain-text message. The procedure is to combine each plain-text character in order with the corresponding character from the pad. The combination is typically performed using modulo addition of the two characters but can be performed with a bitwise XOR. Because characters from the pad may only be used once, the pad must be at least as long as the plain-text message. The one time use of a random string of characters from the pad is what gives the system its perfect security property. If two parties wish to exchange multiple messages, then the pad must be at least as long as the sum of the length of all the messages. The main disadvantage of a one-time pad is that the two parties must each obtain a copy of the same *pad*. This is less of a disadvantage for key recovery because the the encrypted message (keys) does not need to be exchanged with another for decryption but are decrypted by the self-same party so only one copy of the pad is needed.

Suppose for example, a OTP is used to encrypt the key or key history. Given that the adversary does not have access to the OTP then the encryption has perfect secrecy which means that the only viable attack is via brute force. If the encrypted key or key history is at least 128 bits long then brute force is practicaly impossible. Consequently the OTP encrypted key history could be safely stored in a public immutable database. The remaining problem is management of the OTP. Using an OTP to encrypt the key history just creates a new problem, that of securing the OTP itself. But the main advantage of a OTP over secret sharing described above for key recovery is that a OTP approach is non-multi-party interactive. It can be self-contained which is advantageous in data streaming applications.

One common but weaker variant of the OTP is the book cyper. In this variant the OTP is a book. Because the characters in a book are not a random string there is some degree of correlation between characters that makes it less than perfectly secure. Thus two parties who each have a copy of the same book (same edition) can use the characters in the book as the OTP to encrypt messages without ever having to exchange copies of the book. Essentially using a book as OTP is an example of hiding the OTP in plain sight. An adversary would have to guess that a book was being used as a one-time pad and then figure out which book. For key recovery, the key owner merely needs to remember which book and edition. Should the book used by the key owner be lost, the key owner can get another copy from a bookstore.

The book cypher is an interesting example due to the combination of simplicity, the use of existing but readily available sources of information, and the ability to hide the OTP as book in plain sight. This has the advantage that the only the title and edition of a book need to be remembered thus making light demand on human memory. The primary disadvantage of the book cypher is that the text is not random and its difficult to calculate how many bits of entropy are lost for a given book.

Hybrid Key Recovery Method

One of the main attractions of using a one-time pad (OTP) for key recovery, in contrast to secret sharing, is that it is non-multi-party interactive. A hybrid approach that makes a beneficial trade-off is to use a mnemonic merely to generate a seed for a cryptographic strength psuedorandom number generator (CSPRNG). The seed is then used via the CSPRNG to generate a OTP that is then used to encrypt the key-rotation history. The cryptographic strength of the OTP is now governed by the length of the seed not the length of the pad. But key-rotation histories are relatively short compared to the period of CSPRNG so a strong enough seed (128 bits of entrophy) would still be sufficient for this task.

The PRNG algorithm must be of cryptographic quality otherwise it could become a source of vulnerability. A recent advancement in CSPRNG algorithms is a chaotic iteration psuedorandom number generators (CIPRNG). These are of cryptographic quality have extremely high statistical randomness. They pass both the NIST and DieHard tests for PRNG with periods on the order of 109 opt. The basic concept is a chaotic finite state machine cfsm. Unfortunately there do not yet appear to be any open source implementations of this algorithm. A more practial CSPRNG that could be used to generate a OTP from a seed is the libsodium `randombytes_buf_deterministic` function. This uses ChaCha20 under the hood.

The advantage of this hybrid approach is that the key recovery memory task is now limited to merely recovering the seed that would then be used to reproduce the OTP that would then be used in turn to decrypt the key history. This approach does not require multi-party interaction like secret sharing as the seed is directly recovered by the owner via a mnemonic device, not from others. This hybrid approach still benefits from the properties of the OTP for encryption so that the key-rotation history can be encrypted and stored online for recovery.

What remains then is the selection of a mnemonic for generating the seed. It may be difficult for a single mnemonic to provide a random source of seed material at the required strength of 128 bits. Concatenating several sources of mnemonically derived seed material, however, could produce the required strength. This is akin to the DiceWare approach to passphrase generation. One problem with concatenation of seed material is that the order of concatenation must also be remembered. One way to avoid having to remember the order when combining multiple sources of seed material is to use the simple version of secret splitting. In this form of secret splitting, the secret is divided into shards and each shard is XORed together to recover the secret. In this case the secret is the seed and each shard contributes a certain amount of entropy to the final seed. This allows a mnemonic for each shard that may have much less than the required 128 bits of entropy but the combination of shards could have the required entropy and the order of the shards is not important. A non-ordered combination loses some cryptographic strength because the number of possibilties is no longer merely the multiple of the independant possibilities from each shard (permutations) but is instead the number of combinations of the shards.

Suppose that there are four shards that each contribute 35 bits of entropy or in other words each shard is randomly chosen from 235 possibilities. Then the combined number of possibilities is 235 taken four at a time. The exact formula for the combination of N things taken K at a time is given by: `N!/(K!*(N-K)!)` Computing factorials for very large numbers is a computationally intensive task. For the sake of analysis an approximation is sufficient. A lower bound on the number of combinations of *N* things taken *K* at a time is (N/K)K (see bounds). The bits of cryptographic strength of the combination of four shards each with 35 bits is where N = 235 and K = 4. Using the approximation gives the number of possiblities to be at least (235/22)4 = 233*4=2132. This corresponds to 132 bits of entropy, which is greater than the required 128.

The one remaining challenge then is to find good mnemonically recoverable sources of random seed material. One feature that makes the The book cypher was attractive because it took advantage of information that was highly available but hidden in plain sight and whose source was easy to remember (a book title). The problem with books is that the content is not highly random so it in itself is not a good source of seed material. In other words, the challenge is to find sources of information for seed material that have much higher degree entropy than a book but are still easy to remember. More specifically this means finding sources of highly random seed material that are highly available (thus do not require additional infrastructure to backup) but are also essentially hidden in plain sight and easy to recall via a mnemonic device. What follows are several viable sources of mnemonically recoverable sources of random seed material.

DiceWare Seed Recovery

The DiceWare approach can be repurposed to provide a mnemonic source of seed material. These can be used to recover the seed for the one-time pad used to encrypt the key-rotation history. Ten randomly selected words from a DiceWare-compatible wordlist could be used to generate the seed for the one-time pad. Ten randomly selected words in order provide the required 128 bits of entropy (recall that each DiceWare word provides 12.9 bits of entropy). The order of the words is important. Each word would be hashed using SHA-2 or Blake to generate a 16-byte string. The seed is created by concatenating the hashes in the defined order. Once the seed for the OTP is generated, the rest of the recovery method follows the process described above for generating the OTP using a CSPRNG and then using that to encrypt/decrypt the key rotation history. The mnemonic load for this method is the recall the order of ten words from the DiceWare or EFF wordlist. This has a large mnemonic load so it would require some rehearsal and might not be very practical. In addition to the mnemonic at least a physical backup of the ten words should also be created. The physical backup of the ten words could be split into parts to make it more secure. If practical, a multi-party threshold secret sharing backup could also be created.

GitHub Seed Recovery

Github.com stores versioned code repositories. The associated git utility automatically calculates a 160 it (20 byte) SHA-1 hash of each commit to a repository. These hashes are easily readable from the GitHub.com web site. Several Github commit hashes can be used to create the seed to generate the OTP for encrypting the key rotation history. In order to recover a commit hash one must remember the project and repository name, and the date of the commit. If there are multiple commits on the same date then one must also remember which commit, like the last or the first. This is not an onerous memory task but not a trivial one.

There are over 80 million GitHub repositories. A reasonable estimate of the average number of commits per repository is over 1000. This means that there are about 80,000,000 * 1,000 = 80,000,000,000 = 236.22 possibilities to choose from. If a repository/commit is selected randomly then the number of bits of entropy represented by a single choice is about 36. To get 128 bits of security one would need to randomly select four repository/commits. A permutation of 4 gives 4*(36) = 144 bits of entropy. Remembering the order of the four repositories adds another memory task. If instead the four choices were combined using the simple version of secret splitting described above, where each shard is XORed together to recover the secret, then the number of random possibilities is reduced to the number of combinations of 80,000,000,000 items taken four at a time. As previously described, the lower bound on the number of combinations of *N* things taken *K* at a time is (N/K)K. In this case K = 4 and N = 236. This gives the number of possibilities to be (236/22)4 = 234*4=2136. This corresponds to 136 bits of entropy which is still greater than the required 128.

The GitHub.com based recovery mechanism can be summarized as follows: Randomly choose four GitHub.com repository commits. For each commit, the pairing of a project name, repository name and commit date must be remembered and/or backed up using a hardware backup. Generate a seed by XORing together the 20-byte commit SHA-1 commit hash from each of the four repositories. Use this seed with a deterministic CSPRNG to generate a one-time pad of length at least as long as the key rotation history. Encrypt the key rotation history by bitwise XORing each byte in the history with the corresponding byte from the one-time pad. Securely discard the one-time pad. Store the encrypted key-rotation history in a highly available database. This encrypted history should be impervious to attack so it can be stored online. When recovery is required, remember the four project/repository/commit-date pairings or restore from a hardware backup. Use the pairings to lookup the SHA-1 commit hashes from GitHub.com for each. Then recreate the seed by XORing the four commit hashes. Use the seed and the same CRPRNG to regenerate the one-time pad. Retrieve the key history from the database. Use the one-time pad to bitwise XOR each byte of the saved encrypted key history to unencrypt it. The key history is now recovered.

The memory load is four triples of a project name, a repository name, and a date, or twelve items total, but the order of the triples is not important. Given that typically each GitHub project has a small number of repositories, merely remembering the project should make remembering the repository much easier by going to the project page and looking at the choices for repositories. The date is the hardest memory task. There are several well known mnemonic techniques for remembering dates. In addition to the mnemonic, a physical backup of the hashes should also be created. The physical backup could be split into four parts to make it more secure. If practical a threshold multi-party secrete sharing system could provide additional backup.

FlickR.com Seed Recovery

The FlickR.com-based recovery mechanism is similar to the Github.com based one. There are over 10 billion primary photos on FlickR. Each primary photo may come in multiple resolutions. A given photo is displyed on the FlickR.com web page using a low-resolution copy. This displayed version can be scraped from the page. The Flickr.com website does not provide hashes of the images, so one would have to scrape or download an image and then calculate the hash after the fact. A viable approach would be to use SHA-2 from the OpenSSL library or Blake from the libsodium library. Ten billion is about 233.22 which corresponds to about 33 bits of entropy when randomly selected. Four randomly selected images are needed to get the required 128 bits of entropy, that is, 4 * 33 = 132. If we combine the hashes from four images by XORing (i.e. simple secret splitting) then the number of choices becomes the combinations of 10 billion things taken four at a time. As described above, the lower bound on the number of combinations of *N* things taken *K* at a time is (N/K)K. This gives the number of possibilities to be (233/22)4 = 231*4=2124. This corresponds to

124 bits of entropy which is close enough to the required 128. (24 = 16, which is not meaningfully weaker as it would still take 500,000,000 years to break). The proceedure for recovery is essentially the same as the GitHub example above, once the hashes for each photo have been generated.

The mnemonic task is remembering four images. Humans are very good at remembering images given a selection. The hard mnemonic task is searching on FlickR for a given image using tags. It takes about four or five tags to get the list of images to under 100 for a given tag set. The mnemonic task is then to remember four sets of four to five tags each, where the tags are not in any order. Remembering which photo is helped by the fact that the tag set typically corresponds to features of the photo. Moreover, images provide an opportunity to hide them in plain sight. In addition to the mnemonic, a physical backup of the hashes should also be created. The physical backup could be split into four parts to make it more secure. If practical a threshold multi-party secret sharing system could provide additional backup.

Geneological Database Seed Recovery

FamilySearch.org has over six billion genealogical records indexed by name and life-event type, event date, and event place. There are seven standard event types such as birth, death, marriage, census, military service, immigration, and probate. A randomly selected record can be recovered with a name and the event details of event type, date, and place. With six billion records and seven event types there are over 42 billion choices. The number of bits of entropy for one randomly selected record is log2(42,000,000,000) = 35.29. Suppose four records are randomly selected. hTe OTP seed is created by XORing a SHA-2 or Blake hash from each record where the hash is computed from the record name and event details. This produces (235/22)4 = 233*4=2132 combinations which corresponds to 132 bits of entropy. This exceeds the desired 128.

The mnemonic task is to remember the name, event type, event date, and event place for four different records. The records can be in any order. In addition to the mnemonic a physical backup of the hashes should also be created. The physical backup could be split into four parts to make it more secure. If practical a threshold multi-party secrete sharing system could provide additional backup.

Google Maps Seed Recovery

The Google Maps database covers the entire globe with high resolution imagery of the land area. The world's land area is approximately 150,000,000 km2. It has been estimated that 90% of the landmass is inhabited although only 10% is considered urban. Lightly populated areas still have memorable identifiable features suitable for map based mnemonics such as roads, fences, and buildings (farms, huts, etc). The estimated inhabited surface area is 0.9 * 150,000,000 km2 = 135,000,0000 km2.

The resolution of Google Maps' georeferenced satellite photos is given in decimal degrees to six decimal places. For example, clicking on a map gives the location in (degrees latitude, degrees longitude) as (45.348807, -105.709547). Six decimal places is about one tenth of a meter. This is too small to reliably reproduce merely by clicking on the satellite view. Five decimal places is about one meter. This is big enough that it can be reproduced reliably albeit carefully by clicking on the satellite view. A conservative approach would be four decimal places which is about 10 meters. This is easily large enough that it is trivial to reproduce reliably by clicking on the satellite view.

A resolution of approximately one square dekameter (10m)2 or 4 decimal places per location gives a total of 135,000,000 * 10,000 = 1,350,000,000,000 = 240.3 unique locations. When selected randomly this corresponds to over 40 bits of entropy per location. A resolution of a square meter per (1m)2 or 5 decimal places per location gives a total of 135,000,000 * 1,000,000 = 135,000,000,000,000 = 246.94 unique locations. When selected randomly this corresponds to over 46 bits of entropy per location.

At a resolution of a square dekameter four randomly chosen locations are needed to reach over 128 bits of entropy, (4 * 40.3 = 160.9). At a square meter resolution only three randomly chosen locations are needed to reach over 128 bits of entropy, (3 * 46.94 = 140.82.

When locations are combined using a secret splitting approach, the total number of combined unique locations in combination is reduced. As described above, a lower bound on the number of combinations of *N* things taken *K* at a time is (N/K)K. At the square dekameter resolution, K = 4 and N = 240. This gives the number of possibilities to be (240/22)4 = 2384=2152. This corresponds to 152 bits of entropy which is greater than the required 128. At the square meter resolution, K = 3 and N = 246. This gives the number of possibilities to be (246/21.59)3 = 244.413 ~= 2133. This corresponds to 133 bits of entropy which is still greater than the required 128.

Consequently with Google Maps either three or four unique locations are needed to achieve the desired cryptographic strength for seed generation. Memorable locations could include the corner of a building or or a doorway or roofline or road intersection or fenceline intersection or pole. The mnemonic load for a site is the address of the site. Because humans are adept at remembering locations visually by familiarity with the surroundings, exact addresses may not be needed. Merely enough of an address to move the view within the neighborhood of a location may be enough. Once in the neighborhood, terminal navigation may be performed via visual interaction with the Maps app. Alternatively, landmarks, business or other nearby features could be used as the search parameters. In addition the user has to remember what exact feature of the structure is used for the location.

Recovery Summary

All of the hybrid recovery methods allow for rapid recovery that does not require multi-party interaction. They all depend on a non-trival but not onerous mnemonics for rapid recovery but may fall back to a physical or threshold secret sharing multi-party interactive copy for slower recovery. Rapid recovery using the online databases (GitHub.com , FlickR.com, FamilySearch.org, or Google maps) depends on the availability of the databases maintained by the corresponding entities. In each case, should one of the selected records be deleted then the only recourse would be one of the backups.

In order to achieve the required 128 bits of security, the DiceWare approach requires recalling 10 words in order, whereas the GitHub.com, Flickr.com, FamilySearch.org and Google maps (at 1 dekameter) approaches require recalling four records. All five methods could be mixed. Using a mixture adds some security (more choices) but not enough to reduce the number of records required. Alternatively, at one meter resolution the Google maps approach only needs three records. The Google maps approach (either four locations or three locations) may have the lightest memory load because the exploits the high human capacity for visual-geospatial recall.

The secret splitting used to combine records could be augmented to use a threshhold scheme to make it more resilient to record loss but at the cost of needing more than records.

If multi-party interactive recovery is acceptable then using threshold secret sharing could be a better approach. Even when multi-pary interactive is not the preferred approach it could be another backup in addition the a physical backup.

This novel hybrid approach combines multiple cryptographic techiques to provide a viable non-multi-party interactive rapid key recovery method that is well suited to data streaming applications. It combines hiding in plain site, mnemonics, DiceWare-like selection, secret splitting, CSPRNG, and one-time pads. The method is a practical tradeoff between the features of the different approaches.

Virtual World Game as Hierarchically Deterministic Seed Mnemonic

Looking to the future, it would be possible to create a mnemonic-seed generating mobile or desktop application that is completely self-contained and does not require any external online databases for random key material. Humans have an innate ability to remember complex visual geo-spatially related information such as is encountered in everyday life when walking from one place to another without getting lost. Humans are particularly adept at remembering how to retrace the path they followed on a journey through a city, or countryside. Humans are also adept at remembering when the memory is associated with familiar spacial surroundings. The well known method of loci, more commonly known as the memory palace mnemonic, associates a sequence of items to be remembered with locations in one's house or other familiar structure. When a spatial mnemonic is enhanced with what is called [elaborative encoding] (https://en.wikipedia.org/wiki/Elaborative_encoding), that is, adding visual, auditory or other sensory cues, it becomes particularly powerful. Humans are also adept at learning complex mental models via hierarchical decomposition.

Various other mnemonic devices take advantage or combinations of familiar, spatial, hierarchical and sensory cues to make the learning and recall task easier.

An application that exploited multiple mnemonic devices in combination could minimize the memory load required to recover seed material. Indeed games that involve recalling complex sequences of movement and action within a simulated graphical world can be successfully played by young children. This level of mnemonic capability in demonstrated by young children when playing games like The Legend of Zelda. What is being proposed is a hierarchical deterministic seed mnemonic (HDSM) as a type of hierachical spatial elaborative encoded mnemonic.

Lets call this hypothetical mnemonic seed generating game *Quest for the Mnemon Seed* for lack of a better title. A notional description follows: The game is based on a graphical virtual world map such as one might encounter in an online role playing game. In the game, the user starts at the entrance and is presented with a map of a locale such as a village containing unique sites including buildings, parks, roads etc. Each site within the locale has memorably unique visual features such as floor plan, architectural style, period, color, material, flora, fauna, characters, objects etc. The user then walks down roads and paths to get to the different sites. Upon entry to a site the user is presented with a choice of actions to perform such as picking up an object or interacting with a character. Thus the process of selecting a site and then selecting an action at the site constitues a choice. If the choice is selected at random then it becomes the source of random seed material. The mnemonic is remembering where the site is placed within the locale and how to get there and then remember the action(s) performed at the site. A sequence of site visits with actions then provides an extended source of key material. Playing the game provides rehearsal so that a specific set of actions can be recalled in order, thereby recovering the seed.

The site options, both exterior and interior, such as location, layout, style, material, color, etc, are specified as a data structure represented as a sequence of bit fields. A single long string of bytes such as might be generated with a deterministic hash can then be used to generate a uniquely configured locale. A set of sites and actions can also be encoded as a sequence of bit fields. A path through the locale with visits and actions at each site can then be generated from a large random number.

The game is then played in two modes. The first mode generates a random seed and then rehearses the mnemonic for the random seed. The second mode recovers the random seed with the mnemonic.

In the first, generative, mode, the user inputs a string that is the customization phrase. The cryptographic strength of the customization phrase is not important, it just allows the user to have a custom configured locale that is compatible with the user preference. The customization phrase is hashed (with Blake or Sha2) to generate a sequence of bytes used to specify the local options. The locale is then generated. A 2D or 3D display of the locale map is then presented to the user. The game then uses a cryptographic class random number generator to create the 128-bit random seed. This seed will be used to generate the one-time pad for encrypting the key-rotation history. Using the seed and a CSPRNG, a sequence of sites and actions is created deterministically from the seed as the mnemonic. The user is then shown on the map this mnemonic path through the locale. The user follows the path through the locale, visiting each site in turn, where the user is prompted to perform the selected action or actions. Once complete the user continues to rehearse the mnemonic, only now the path is not shown. The user must recall it from memory. If the user makes a wrong choice, the game reminds the user with a prompt. Rehearsal repeats until the user can successfully retrace the path and actions from memory without any prompts. At this point the user has memorized the mnemonic and can print out copies of the random seed for backup, use it for generating a one-time encryption pad, and then instruct the application to forget the random seed.

In the second, recovery, mode, the user inputs the customization phrase to generate the locale map. The user then visits sites in turn and performs actions at each site. The sequence of site visits and actions deterministically regenerates a seed. When the user completes a sequence the game displays the associated seed. If the user correctly replayed the sequence then the user will recover the correct seed. If the user does not, then the seed provided by the game will not be the one the user was trying to recover.

Suppose that each locale contains 256 = 28 sites. This is comparable to a small village of population about 1000. Randomly selecting a site then provides 8 bits of entropy. Suppose that inside each site there are 8 = 23 spots, such as cupboard east wall, shelf north wall, barrel northeast corner, etc. Random selection of a spot would provide 3 more bits of entropy. Suppose that at each interior location the user has 2 = 21 choices of action such as, pick up hammer, drink vial of liquid, answer question from inn keeper, etc. Random selection of an action would provide another 1 bit

of entropy. Suppose then that after completing the first spot-action the user has to select another spot and make another binary choice of action. The second spot-action provides yet another 4 bits of entropy. This given a total of 8 + 4 + 4 = 16 bits of entropy per site-spot-action-spot-action sequence. To provide the total of 128 = 8 * 16 bits of entropy needed for the random seed requires that the user visits 8 sites in order while selecting two successive actions at each site.

Alternatively the game could provide some other mix of interior location and interaction choices to get 8 bits of entropy. Suppose for example that at each of the 256 sites there are 32 = 25 spots. Random selection of a spot provides 5 bits of entropy. At each spot there are 8 = 23 action choices. Random action selection provides another 3 bits of entropy. So each spot-action selection provides 5 + 3 = 8 bits of entropy. If at each site the user must make 3 spot-action selections then that provides a total of 3*8 = 24 bits of entropy. Thus each site-spot-action-spot-action-spot-action combintion or site + (spot-action) * 3 combination provides 32 = 8 + (3 * 8) bits of entropy. A 128 = 4 * 32 bit seed can then be generated from only four site-(spot-action)3 combinations, that is, 128 = 4 * (8 + (3 * 8)). An area of research would be to find the optimal decomposition and combination of site-spot-action sequences.

Either of the eight-site or four-site examples above are well within the mnemonic capabilities of the general population given the dense hierarchical geospatial sensory cues that such a graphical virtual game world journey provides and would only take a few minutes to replay for recovery. The app would run self contained on the user's mobile device or desktop computer and would make seed recovery fun. Any computing device could be engaged to play the app so it would not require a specific mobile device or computer and therefore loss of the user's mobile device would not impede seed recovery.

A variation of the game would be to allows some sites to have a portal that transports the user to a new locale with a new unique map. The configuration of the new locale is determined by a hash of the site/action visit selections that were performed prior to entry of the portal. This would add additional variety to the game and help differentiate the mnemonics required for the create of multiple unique seeds. This makes the game a recursively hierarchical deterministic seed mnemonic (RHDSM).

This hierarchically deterministic seed mnuemonic (HDSM) could become a standard feature for primary key recovery for any decentralized identity based cryptographic system where the user must generate and manage their private keys. Once users become familiar with this approach to key recovery it could open the door to more rapid adoption of decentralized approaches to online interactions where security is based on user managed public/private key pairs.

3.3.4 Summary

A new data type called a DAD for decentralized autonomic data has been presented that is derived from decentralized identifiers, DIDs. DADs are suitable for streaming applications. Methods for the three basic key management operations, namely, reproduction, rotation, and recovery have been presented that are compatible with DAD stream-data applications. The pre-rotation and hybrid recovery methods presented in this paper including the hierarchically deterministic seed mnuemonic (HDSM) are somewhat novel. They all provide what could be considered minimally sufficient means for key management operations.

3.4 Appendices

3.4.1 Support for DAD Signatures in HTTP

In web applications that use HTTP, the simplest most compatible way to associate or attach a signature to an HTTP packet is to include it in a custom HTTP header. Standad JSON parsers raise an error if there are additional characters after a closing object bracket thus one cannot simply append the signature after the JSON serialization in the message body. Another approach would be to use a custom JSON parser that guarantees a cononical representation of a JSON serialization (including white space) and then wrap the data item and the signature in another JSON object, where the signature and the data item are both field in the wrapper object. This is more verbose and is not compatible with the vast majority of web application framework tools for handling JSON serialized message bodies. Thus it is non-trivial

to include the signature in the message body. Using a custome HTTP header is relatively easy and has the advantage that is is compatible with the vast majority of existing web frameworks.

A suggested header name is *Signature* header that provides one or more signatures of the request/response body text.

The format of the custom Signature header follows the conventions of RFC 7230

Signature header has format:

```
Signature: headervalue

Headervalue:
  tag = "signature"
or
  tag = "signature"; tag = "signature"  ...

where tag is replaced with a unique string for each signature value
```

An example is shown below where one *tag* is the string *signer* and the other *tag* is the string *current*.

```
Signature: signer="Y5xTb0_jTzZYrf5SSEK2f3LSLwIwhOX7GEj6YfRWmGViKAesa08UkNWukUkPGuKuu-
↪EAH5U-sdFPPboBAsjRBw=="; current="Xhh6WWGJGgjU5V-e57gj4HcJ87LLOhQr2Sqg5VToTSg-
↪SI1W3A8lgISxOjAI5pa2qnonyz3tpGvC2cmf1VTpBg=="
```

Where tag is the name of a field in the body of the request whose value is a DID from which the public key for the signature can be obtained. If the same tag appears multiple times then only the last occurrence is used.

Each signature value is a doubly quoted string `""` that contains the actual signature in Base64 url safe format. But the signatures should use an intelligent default cryptographic suite such as 64-byte Ed25519 signatures that have been encoded into BASE64 url-file safe format. The encoded signatures are 88 characters in length and include two trailing pad characters =.

An optional *tag* name = *kind* may be present to specify the cryptographic suite and version of the signatures. The *kind* tag field value specifies the type of signature. All signatures within the header must be of the same kind.

```
Signature: signer="B0Qc72RP5IOodsQRQ_
↪s4MKMNe0PIAqwjKsBl4b6lK9co2XPZHLmzQFHWzjA2PvxWso09cEkEHIeet5pjFhLUDg==";  did=
↪"B0Qc72RP5IOodsQRQ_
↪s4MKMNe0PIAqwjKsBl4b6lK9co2XPZHLmzQFHWzjA2PvxWso09cEkEHIeet5pjFhLUDg==";  kind=
↪"ed25519:1.0"
```

3.4.2 Cryptographic Suite Representation

Best practices cryptography limits the options that user may choose from for the various cryptographic operations, such as signing, encrypting, and hashing to a suite of balanced and tuned set of protocols, one for each operation. Each member of the set should be the one and only one best suited to that operation. This prevents the user from making bad choices. In most key-representation schemes each operation is completely free to be specified independent of the others. This is a very bad idea. Users should not be custom combining different protocols that are not part of a best practices cypher suite. Each custom configuration may be vunerable to potential attack vectors for exploit. The suggested approach is to specify a cypher suite with a version. If an exploit is discovered for a member of a suite and then fixed, the suite is updated totally to a new version. The number of cypher suites should be minimized to those essential for compatibility but no more. This approach increases expressive power because only one element is needed to specify a whole suite of operations instead of a different element per operation.

See this article for a detailed explanation on how standards such as JOSE expose vulnerabilities due to too much flexibility in how cryptographic operations are specified.

Example cypher suites:

```
v1: Ed25519, X25519, XSalsa20poly1305, HMAC-SHA-512-256
v2: Ed448, X448, XChaCha20Poly1305, keyed BLAKE2b
v3: SPHINCS-256, SIDH, NORX64-4-1, keyed BLAKE2x
```

3.4.3 Canonical Data Serialization

Canonical data serialization means that there is a universally defined way of serializing the data that is to be cyptographically signed.

The are few typical approaches to achieving data canonicalization. The advantages of compatibility, flexibility, and modularity that come from using a key/value store serialization such as JSON usually makes 1) the preferred approach.

1. Store the serialization and signature as a chunk.

The simplest is that the signer is the only entity that actually serializes the data. All other users of the data only deserialize. This simplifies the work to guarantee canonization. For example, JSON is the typical data format used to serialize key:value or structured data. But the JSON specifcation for ser/deser treats whitespace characters and the order of appearance of keys as semantically unimportant. For a dictionary (key:value) data structure the typical approach is to represent it internally as a hash table. Most hash algorithms do not store data ordered in any predictable way (Python and other languages have support for Ordered Dicts or Ordered Hashes, which can be used to partially ameliorate this problem). But from the perspective of equivalence, key:value data structures are "dict" equal if they have the same set of keys with the same values for each key. Thus deserialization can produce uniform equivalent "dict equal" results from multiple but differing serializations (that differ in whitespace and order of appearance of fields). JSON only guarantees *dict* equivalent not serialization equivalence. Unfortunately the signatures for the differing but equivalent serializations will not match.

But in signed at rest data only the signer ever needs to serialize the data. Indeed, only the signer may serialize the data because only the signer has the private key. So deserialization and reserialization by others is of limited value. The primary value appears to be either schema completeness where signatures are included as fields in a wrapper object or the ability to nest signatures or signed data with signatures. Because it is simple to convert a JSON serialization to a coded serializaiton such as Base64, nested coded JSON serialization without canonicalization can be trivially supported. After expansion and decoding, readers of the data can see the uncoded underlying data in a *schema complete* representation.

The signer's serialization is always *canonical* for the signature. Users of the data merely need to use a "dict equal" deserialization which is provided by any compliant JSON deserializer. So no additional work is required to support it across multiple languages etc. If the associated data also needs to be stored, unserialized then validation and extraction of the data is performed by first verifying the signature on the stored serialization and then deserializing it in memory.

2. Implement perfectly canonical universally reproducibly serialization.

In this approach all implementations of the protocol or service use the exact same serialization method that is canonical including white space and field order so that they can reproduce the exact same serialization that the original signer created when originally signing the data. This is difficult to achieve with something like JSON across multiple languages, platforms, and tool kits. It's usually more work to implement and more work to support because it usually means either using something other than JSON for serialization or writing from scratch conformant JSON implementations or at the very least having tight control of how white space and order occurs and ensuring accross updates that this does not change. Unfortunately many overly schematizied standards are based on this approach. This approach typically breaks web application frameworks.

3. Use binary data structures

With binary data structures the canonical form is well defined but it is also highly inflexible.

3.4.4 Relative Expressive Power

One way to measure and compare different knowledge representations is called *relative expressive power*. In the physics world *power* is defined as work done per unit time. It is a ratio. *Expressive power* is similary defined as the ratio of meaning conveyed per dependency, where dependency is something that must be kept track of or transmitted to convey the meaningful information. Because dependencies are a measure of complexity, relatively higher expressive power conveys more meaning relatively more simply.

Intelligent Defaults

One approach to acheiving higher expressive power in a data representation specification is the use of intelligent defaults. An intelligent default assigns meaning to the absence of data. For example, if there are several options for a given data item value such as the *type* of a data item, an intelligent default would assign the type to a predetermined default if no type is provided in the data. This provides high expressive power as the type meaning is conveyed without the transmission of any bytes to represent type.

Typically in any given knowledge representation application the relative frequency of the appearance of optional values is not evenly distributed, but follows a Pareto distribution. This means that if an intelligent default (the Pareto optimal value) is specified as part of the schema the average expressive power of data items will be increased.

A practical example of this is the RAET (Reliable Asynchronous Event Transport) protocol header (see RAET). Typically in protocols the header has a fixed format binary representation for two reasons. The first is that every packet includes the header, so a verbose header reduces the payload capacity of each packet, thereby making the protocol comsume more bandwidth. The second is that the header is used to interpret the rest of the packet and therefore must be consistenly parsable which is easier if the format is fixed. The problem with fixed format headers is that they are not extensible. To make the extensible usually means adding additional fields to the header to indicate the presence of additional extended fields. RAET used an *intelligent default policy* to achieve a completely flexible extensible header that on average is the size of a non-extensible fixed format header. In RAET the header is composed of a serialized list of key-value pairs where each key is the field name of the associated field value. This makes it easy to add new key-value pairs as needed to extend the protocol to different uses and with different behavior. Unfortunately, transmitting the keys makes the header much larger relative to a fixed format header where the offset of the value in the header determines the associated field. RAET overcomes this problem by defining a default value for each key-value pair. When a header is generated on the transmit side, the actual key-value pairs are compared against the default set. Any pair where the value matches the default is not included in the list of key-value pairs in the transmitted header. On the recieve side a default header is created with every key value pair set to the default. The received header's key-value pairs are used to update the default header with the non-defaulted values. Because the optional fields are seldomly used by most packets the average header size is comparable to a fixed format header. When viewing the header after expansion and update, all the fields are present, so there is no hidden information. All the meaning is apparently conveyed.

RAET header field defaults

```
PACKET_DEFAULTS = odict([
                              ('sh', DEFAULT_SRC_HOST),
                              ('sp', RAET_PORT),
                              ('dh', DEFAULT_DST_HOST),
                              ('dp', RAET_PORT),
                              ('ri', 'RAET'),
                              ('vn', 0),
                              ('pk', 0),
                              ('pl', 0),
                              ('hk', 0),
                              ('hl', 0),
                              ('se', 0),
                              ('de', 0),
```

(continues on next page)

(continued from previous page)

```
                        ('cf', False),
                        ('bf', False),
                        ('nf', False),
                        ('df', False),
                        ('vf', False),
                        ('si', 0),
                        ('ti', 0),
                        ('tk', 0),
                        ('dt', 0),
                        ('oi', 0),
                        ('wf', False),
                        ('sn', 0),
                        ('sc', 1),
                        ('ml', 0),
                        ('sf', False),
                        ('af', False),
                        ('bk', 0),
                        ('ck', 0),
                        ('fk', 0),
                        ('fl', 0),
                        ('fg', '00'),
                ])
```

Any key-value based schema standard specification may benefit from an intelligent default policy to greatly increase the expressive power of the schema. This becomes even more important where security is concerned as the intelligent default might be the most secure set of options thus helping the user be more secure and more expressive. Moreover expressive power is about conveying meaning more simply which makes it easier to implement and incentivizes adoption.

Essential vs. Optional Elements

Another related technique for increasing expressive power is to distinguish between essential and optional elements in a given representation. Any essential elements should be expressed as explicitly as possible (when not defaulted); that is, it should not be looked up and should either not be indirected or have minimal indirection. External lookups are expensive. Moreover, hiding essential elements behind multiple levels of indirection may make it harder to understand the conveyed meaning (adding dependencies and hence complexity). An important meaningful difference that should be apparent is whenever an essential element is not set to a default value. This difference should not be hidden behind indirection.