winternitz Documentation

Release unknown

Harald Heckmann

Mar 11, 2019

Contents

This is the documentation of **winternitz**.

Note: Welcome to the documentation of the python winternitz package. The package contains one-time-signature schemes, which are most likely post-quantum secure.

CHAPTER 1

Contents

1.1 License

1.2 Contributors

- Harald Heckmann <harald.heckmann93@web.de>

1.3 Changelog

1.3.1 Pre Version 1.0

- Setup project (structure, travis, tox, coverage, sphinx, git-prehooks)

1.3.2 Version 1.0

- First fully tested and documented release of the winternitz package.
- Contains AbstractOTS base class for OTS implementations in this package
- Contains fully configurable Winternitz One-Time-Signature scheme
- Contains fully configurable Winternitz One-Time-Signature+ scheme

1.4 Introduction

Lamport invented an algorithm in 1979 which allowed one to create one-time-signatures using a cryptographically secure one-way function. It is the basis for the Winternitz one-time-signature algorithm. Winternitz added the possibility to adjust the tradeoff between time- and space-complexity.

1.4.1 Lamport one-time-signature scheme

Lamport suggested to create two secret keys for each bit of a message which will be signed. One for each value the bit can take. To derive the verification key, each secret key is hashed once. Now you have a secret key and a verification key, which consists of m 2-tuples of values, where m is the number of bits of the message. The verification key is published. The signature consists of m values. For each bit of the message you release a secret key from the corresponding secret keys, depending on which value the bit has. All those secret keys form the signature for the message. The verifier hashes each of your secret keys once and compares it to one verification key for this position, depending on the value of the bit. The signature is valid, if and only if all derived verification keys match with your published verification key at the correct position of the 2-tuple, which is determined by the value of the bit. This algorithm is quite fast (comparing it to existing PQC-algorithms), but the signature sizes are huge.

1.4.2 Winternitz extension

Winternitz extended lamports algorithm by offering the possiblity to decide how many bits will be signed together. The amount of numbers those bits can represent is called the Winternitz parameter ($w = 2^{bits}$). This method offers the huge advantage that the user of this algorithm can choose the time and space tradeoff (whether speed or storage capacity is more relevant). A fingerprint of the message which will be signed is split into groups of $ceil(log_2(w))$ bits. Each of these groups gets one secret key. Each verification key is derived by hashing the secret key for each group 2^{w-1} times. All verification keys will be published and represent one unified verification key. When signing a message, the fingerprint of the message is split into groups of $ceil(log2(w))$ bits. To create the signature, the private key for each bit group is hashed $bitgroup\_value$ times, where $bitgroup\_value$ is the value of the bitgroup. Additionally a (inverse sum) checksum is appended, which denies man-in-the-middle attacks. The checksum is calculated from the signature, split into bit groups of $ceil(log2(w))$ bits, and signed. To verify the signature, the fingerprint of the message is first split into bit groups of $ceil(log2(w)$ bits each. The basic idea is to take the signature of each bit group, calculate the verification key from it and finally compare it to the published verification key. Since the signature was hashed $bitgroup\_value$ times, all you have to do to calculate the verification key from the signature is to hash the signature $2^{w-1} - bitgroup\_value - 1$ times. Besides verifing the message, the verifier must also calculate the checksum and verify it.

1.5 Setup

Requires: Python >= 3.4

Install package: `pip install winternitz`
Install test tools: `pip install winternitz[TEST]`
Install linter (for tox tests): `pip install winternitz[LINT]`
Install documentation tools: `pip install winternitz[DOCS]`
Install everything: `pip install winternitz[ALL]`

1.5.1 Test

Without tox (no linter checks): `python setup.py test`
With tox: `python -m tox`

1.5.2 Generate documentation

```
python setup.py docs
```

1.6 Usage

The package *winternitz* contains a module called *signatures*. Within this package you can find the classes WOTS and WOTSPLUS. Those classes can be used out of the box to sign or verify messages

1.6.1 WOTS

```
import winternitz.signatures
# Create signature and verify it with the same object
wots = winternitz.signatures.WOTS()
message = "My message in bytes format".encode("utf-8")
sig = wots.sign(message)
success = wots.verify(message=message, signature=sig["signature"])
print("Verification success: " + str(success))
# Output: Verification success: True
```

If you don't specify any values in the constructor of WOTS, it will use the winternitz parameter 16 and the hash function *sha512* as default parameters. The private key will be generated from entropy. After you received the public key, either through `wots.pubkey` or inside the dict that is returned by the `wots.sign(message)` function call, you publish it. Verify that it was not modified. In the best case a man-in-the-middle attack to modify your public key is impossible by the design of the application. The last step is to publish your message and every information in the dict that is returned by `wots.sign(message)`, except the public key (since it was already published). Publishing the fingerprint is optional, since it is not essential for the signature verification. The signature dict contains the following values:

```
{
    "w":            winternitz parameter (Type: int),
    "fingerprint":  message hash (Type: bytes),
    "hashalgo":     hash algorithm (Type: str),
    "digestsize":   hash byte count (Type: int),
    "pubkey":       public key (Type: List[bytes]),
    "signature":    signature (Type: List[bytes])
}
```

With that data, another person can verify the authenticity of your message:

```
# Another person or machine wants to verify your signature:
# get required hash function by comparing the name
# published with local implementaitons
if sig["hashalgo"] == "openssl_sha512":
    hashfunc = winternitz.signatures.openssl_sha512
elif sig["hashalgo"] == "openssl_sha256":
    hashfunc = winternitz.signautres.openssl_sha256
else:
    raise NotImplementedError("Hash function not implemented")

wots_other = winternitz.signatures.WOTS(w=sig["w"], hashfunction=hashfunc,
                                        digestsize=sig["digestsize"], pubkey=sig[
↪"pubkey"])
success = wots_other.verify(message=message, signature=sig["signature"])
print("Verification success: " + str(success))
# Output: Verification success: True
```

1.6.2 WOTSPLUS

```
import winternitz.signatures
wotsplus = winternitz.signatures.WOTSPLUS()
message = "My message in bytes format".encode("utf-8")
sig = wotsplus.sign(message)
success = wotsplus.verify(message=message, signature=sig["signature"])
print("Verification success: " + str(success))
# Output: Verification success: True
```

If you don't specify any values in the constructor of WOTSPLUS, it will use the winternitz parameter 16 and the hash function defaults to *sha256*. It further requires a pseudo random function, which defaults to *HMAC-sha256*, as well as a seed which is also generated from entropy. For further informations about functions and their parameters, visit the module reference in this documentation. Since WOTS+ uses a pseudo random function and a seed to derive signatures and public keys, they have to be published as well. In addition to the signature of WOTS, the returned dict contains the following values:

```
{
    # ...
    "prf":          pseudo random function (Type: str),
    "seed":         Seed used in prf (Type: bytes)
}
```

Those arguments have to be specified in the constructor of WOTSPLUS in addition to those parameters specified in WOTS.

1.6.3 Misc

The WOTS classes come with some features that will be explained in the following sections.

Fully configurable

The WOTS classes are fully parameterizable. You can specify anything that is specified in the papers describing the algorithm, including the Winternitz parameter, the hash function, the pseudo random function (WOTSPLUS), the seed (WOTSPLUS), the private key and the public key. specifing both a private key and public key results in the public key beeing discarded.

On-demand generation of keys

If no private key or no public key is specified, they will be set to None. The same goes for the seed in wots+. Only when they are required, they will be generated or derived. This means that as long as you don't execute `repr(obj)`, `str(obj)`, `obj1 == obj2`, `obj1 != obj2`, `obj.pubkey`, `obj.privkey`, `obj.sign(...)` or `obj.verify(...)`, where obj is a WOTS object, the keys will stay None.

Code representation of WOTS objects

You can call `repr(obj)`, where obj is a WOTS object, to get a line of code which contains all information to initialize another object so that it is equal to obj. Executing `obj2 = eval(repr(obj))` executes that code which is returned by `repr(obj)` and ultimately stores a copy of it in `obj2`.

Human readable string representation

You can call `str(obj)` to get a string which contains a human readable representation of that object.

Comparison of objects

You can compare two objects from this class `obj1 == obj2` and `obj1 != obj2`

Optimizations

The code was carefully written to reduce execution times. It surely is not perfect and can still be optimized, further time-critical sections could be coded as C extensions, but nevertheless in the current state it should offer quite an efficient implementation. It defines `__slots__` to reduce execution times and storage requirements within the class. Implementation of parallelization is planned, but it is only usefull when using huge winternitz parameters, since python can only execute code in parallel if you spawn a new process and the overhead of forking a new python interpreter is not negliable.

1.7 winternitz

1.7.1 winternitz package

Submodules

winternitz.signatures module

class winternitz.signatures.**AbstractOTS**

Bases: object

OTS base class

Every class implementing OTS schemes in this package should implement the functions defined in this base class

sign() → dict

Sign a message

This function will create a valid signature for a message on success

Parameters **message** – Encoded message to sign

Returns

A dictionary containing the fingerprint of the message, which was created using the hash function that was specified during initialization of this object, the signature and a public key to verify the signature. Structure:

```
{
    "w":            winternitz parameter (Type: int),
    "fingerprint":  message hash (Type: bytes),
    "hashalgo":     hash algorithm (Type: str),
    "digestsize":   hash byte count (Type: int),
    "pubkey":       public key (Type: List[bytes]),
    "signature":    signature (Type: List[bytes])
}
```

verify(*signature: List[bytes]*) → bool

Verify a message

Verify whether a signature is valid for a message

Parameters

- **message** – Encoded message to verify
- **signature** – Signature that will be used to verify the message

Returns Whether the verification succeded

class winternitz.signatures.**WOTS**(*w: int = 16*, *hashfunction: Callable = <function openssl_sha512>*, *digestsize: int = 512*, *privkey: Optional[List[bytes]] = None*, *pubkey: Optional[List[bytes]] = None*)

Bases: *winternitz.signatures.AbstractOTS*

Winternitz One-Time-Signature

Fully configurable class in regards to Winternitz paramter, hash function, private key and public key

__init__(*w: int = 16*, *hashfunction: Callable = <function openssl_sha512>*, *digestsize: int = 512*, *privkey: Optional[List[bytes]] = None*, *pubkey: Optional[List[bytes]] = None*) → None

Initialize WOTS object

Define the parameters required to sign and verify a message

Parameters

- **w** – The Winternitz parameter. A higher value reduces the space complexity, but increases the time complexity. It must be greater than 1 but less or equal than $2^{digestsize}$. To get the best space to time complexity ratio, choose a value that is a power of two.
- **hashfunction** – The hashfunction which will be used to derive signatures and public keys. Specify a function which takes bytes as an argument and returns bytes that represent the hash.
- **digestsize** – The number of bits that will be emitted by the specified hash function.
- **privkey** – The private key to be used for signing operations. Leave None if it should be generated. In this case it will be generated when it is required.
- **pubkey** – The public key to be used for verifying signatures. Do not specify it if a private key was specified or if it should be derived. It will be derived when it is required.

digestsize
Digest size getter

Get the digest size of the hash function

Returns Digest size of the hash function

hashfunction
Hash function getter

Get a reference to the current hash function

Returns Reference to hash function

privkey
Private key getter

Get a copy of the private key

Returns Copy of the private key

pubkey
Public key getter

Get a copy of the public key

Returns Copy of the public key

sign(*message: bytes*) → dict
Sign a message

This function will create a valid signature for a message on success

Parameters **message** – Encoded message to sign

Returns

A dictionary containing the fingerprint of the message, which was created using the hash function that was specified during initialization of this object, the signature and a public key to verify the signature. Structure:

```
{
    "w":            winternitz parameter (Type: int),
    "fingerprint":  message hash (Type: bytes),
    "hashalgo":     hash algorithm (Type: str),
    "digestsize":   hash byte count (Type: int),
    "pubkey":       public key (Type: List[bytes]),
    "signature":    signature (Type: List[bytes])
}
```

slots = ['__weakref__', '__w', '__hashfunction', '__digestsize', '__privkey', '__pubke

verify(*message: bytes, signature: List[bytes]*) → bool
Verify a message

Verify whether a signature is valid for a message

Parameters

- **message** – Encoded message to verify
- **signature** – Signature that will be used to verify the message

Returns Whether the verification succeded

w
Winternitz parameter getter

Get the Winternitz parameter

Returns Winternitz parameter

class winternitz.signatures.**WOTSPLUS**(*w: int = 16*, *hashfunction: Callable = <function openssl_sha256>*, *prf: Callable = <function hmac_openssl_sha256>*, *digestsize: int = 256*, *seed: Optional[bytes] = None*, *privkey: Optional[List[bytes]] = None*, *pubkey: Optional[List[bytes]] = None*)

Bases: *winternitz.signatures.WOTS*

Winternitz One-Time-Signature Plus

Fully configurable class in regards to Winternitz paramter, hash function, pseudo random function, seed, private key and public key

__init__(*w: int = 16*, *hashfunction: Callable = <function openssl_sha256>*, *prf: Callable = <function hmac_openssl_sha256>*, *digestsize: int = 256*, *seed: Optional[bytes] = None*, *privkey: Optional[List[bytes]] = None*, *pubkey: Optional[List[bytes]] = None*)
Initialize WOTS object

Define under which circumstances a message should be signed or verified

Parameters

- **w** – The Winternitz parameter. A higher value reduces the space complexity, but increases the time complexity. It must be greater than 1 but less than :math: 2^{digestsize}. To get the best space to time complexity ratio, choose a value that is a power of two.
- **hashfunction** – The hashfunction which will be used to derive signatures and public keys. Specify a function which takes bytes as an argument and returns bytes that represent the hash.
- **digestsize** – The number of bits that will be emitted by the specified hash function.
- **privkey** – The private key to be used for signing operations. Leave None if it should be generated. In this case it will be generated when it is required.
- **pubkey** – The public key to be used for verifying signatures. Do not specify it if a private key was specified or if it should be derived. It will be derived when it is required.
- **seed** – Seed which is used in the pseudo random function to generate bitmasks.
- **prf** – Pseudo random function which is used to generate the bitmasks.

prf
Pseudo random function getter

Get the pseudo random function. It is used to generate the bitmasks.

Returns Reference to the pseudo random function

seed
Seed getter

Get the seed which is used in the pseudo random function to generate the bitmasks.

Returns Seed for pseudo random function

sign(*message: bytes*) → dict
Sign a message

This function will create a valid signature for a message on success

Parameters **message** – Encoded message to sign

Returns

A dictionary containing the fingerprint of the message, which was created using the hash function that was specified during initialization of this object, the signature and a public key to verify the signature. Structure:

```
{
    "w":            winternitz parameter (Type: int),
    "fingerprint":  message hash (Type: bytes),
    "hashalgo":     hash algorithm (Type: str),
    "digestsize":   hash byte count (Type: int),
    "pubkey":       public key (Type: List[bytes]),
    "prf":          pseudo random function (Type: str),
    "seed":         Seed used in prf (Type: bytes),
    "signature":    signature (Type: List[bytes])
}
```

slots = ['__weakref__', '__seed', '__prf']

verify(*message: bytes, signature: List[bytes]*) → bool
Verify a message

Verify whether a signature is valid for a message

Parameters

- **message** – Encoded message to verify
- **signature** – Signature that will be used to verify the message

Returns Whether the verification succeded

winternitz.signatures.**hmac_openssl_sha256**(*message: bytes*, *key: bytes*) → bytes
Peudo random function for key and bitmask generation

This functions wraps a pseudo random function in a way that it takes a byte-sequence as an argument and returns a value which can be used for further generation of keys.

Parameters

- **message** – Byte-sequence to be hashed
- **key** – key to be used

Returns HMAC-sha256 hash

winternitz.signatures.**openssl_sha256**(*message: bytes*) → bytes
Hash function for signature and public key generation

This functions wraps a hashfunction in a way that it takes a byte-sequence as an argument and returns the hash of that byte-sequence

Parameters **message** – Byte-sequence to be hashed

Returns Sha256 hash

winternitz.signatures.**openssl_sha512**(*message: bytes*) → bytes
Hash function for signature and public key generation

This functions wraps a hashfunction in a way that it takes a byte-sequence as an argument and returns the hash of that byte-sequence

Parameters **message** – Byte-sequence to be hashed

Returns Sha512 hash

Module contents

1.8 Contribution

This is an open-source project which was created to learn and to have fast and easy access to winternitz signature schemes as a python developer. This project can be optimized and extended, but alone this is quite a difficult task. If you want to contribute, feel free to create an issue or a pull request. If you plan to put more than a couple of hours into extending this package, contact me please before you begin to work on it harald.heckmann93@web.de. In case of implementing new signature schemes, make sure that your OTS class does inherit from winternitz.signatures.AbstractOTS.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

W

Index

Symbols

A

D

H

O

P

S

V

W