argon2$_{c}ffiDocumentation$

Release 18.1.0

Hynek Schlawack

Aug 30, 2018

Contents

Release v18.1.0 (*What's new?*).

Argon2 won the Password Hashing Competition and `argon2_cffi` is the simplest way to use it in Python and PyPy:

```
>>> from argon2 import PasswordHasher
>>> ph = PasswordHasher()
>>> hash = ph.hash("s3kr3tp4ssw0rd")
>>> hash
'$argon2i$v=19$m=512,t=2,p=2$5VtWOO3cGWYQHEMaYGbsfQ$AcmqasQgW/wI6wAHAMk4aQ'
>>> ph.verify(hash, "s3kr3tp4ssw0rd")
True
>>> ph.verify(hash, "t0t411ywr0ng")
Traceback (most recent call last):
  ...
argon2.exceptions.VerifyMismatchError: The password does not match the supplied hash
```

Note: passlib 1.7.0 and later offers Argon2 support using this library too.

`argon2_cffi`'s documentation lives at Read the Docs, the code on GitHub. It's rigorously tested on Python 2.7, 3.4+, and PyPy.

CHAPTER 1

User's Guide

1.1 Argon2

Note: **TL;DR**: Use *argon2.PasswordHasher* with its default parameters to securely hash your passwords.

You do **not** need to read or understand anything below this box.

Argon2 is a secure password hashing algorithm. It is designed to have both a configurable runtime as well as memory consumption.

This means that you can decide how long it takes to hash a password and how much memory is required.

Argon2 comes in three variants:

Argon2d is faster and uses data-depending memory access, which makes it less suitable for hashing secrets and more suitable for cryptocurrencies and applications with no threats from side-channel timing attacks.

Argon2i uses data-independent memory access, which is preferred for password hashing and password-based key derivation. Argon2i is slower as it makes more passes over the memory to protect from tradeoff attacks.

Argon2id is a hybrid of Argon2i and Argon2d, using a combination of data-depending and data-independent memory accesses, which gives some of Argon2i's resistance to side-channel cache timing attacks and much of Argon2d's resistance to GPU cracking attacks.

1.1.1 Why "just use bcrypt" Is Not the Best Answer (Anymore)

The current workhorses of password hashing are unquestionably bcrypt and PBKDF2. And while they're still fine to use, the password cracking community embraced new technologies like GPUs and ASICs to crack password in a highly parallel fashion.

An effective measure against extreme parallelism proved making computation of password hashes also *memory* hard. The best known implementation of that approach is to date scrypt. However according to the Argon2 paper, page 2:

> [. . .] the existence of a trivial time-memory tradeoff allows compact implementations with the same energy cost.

Therefore a new algorithm was needed. This time future-proof and with committee-vetting instead of single implementors.

1.1.2 Password Hashing Competition

The Password Hashing Competition took place between 2012 and 2015 to find a new, secure, and future-proof password hashing algorithm. Previously the NIST was in charge but after certain events and revelations their integrity has been put into question by the general public. So a group of independent cryptographers and security researchers came together.

In the end, Argon2 was announced as the winner.

1.2 Installation

1.2.1 Using the Vendored Argon2

```
pip install argon2_cffi
```

should be all it takes.

But since `argon2_cffi` vendors Argon2's C code by default, it can lead to complications depending on the platform.

The C code is known to compile and work on all common platforms (including x86, ARM, and PPC). On x86, an SSE2-optimized version is used.

If something goes wrong, please try to update your `cffi`, `pip` and `setuptools` first:

```
pip install -U cffi pip setuptools
```

Overall this should be the safest bet because `argon2_cffi` has been specifically tested against the vendored version.

Wheels

Binary wheels for macOS, Windows, and Linux are provided on PyPI. With a recent-enough `pip` and `setuptools`, they should be used automatically.

Source Distribution

A working C compiler and CFFI environment are required. If you've been able to compile Python CFFI extensions before, `argon2_cffi` should install without any problems.

1.2.2 Using a System-wide Installation of Argon2

If you set `ARGON2_CFFI_USE_SYSTEM` to `1` (and *only* `1`), `argon2_cffi` will not build its bindings. However binary wheels are preerred by `pip` and Argon2 gets installed along with `argon2_cffi` anyway.

Therefore you also have to instruct `pip` to use a source distribution:

```
env ARGON2_CFFI_USE_SYSTEM=1 \
  pip install --no-binary=argon2_cffi argon2_cffi
```

This approach can lead to problems around your build chain and you can run into incompatabilities between Argon2 and `argon2_cffi` if the latter has been tested against a different version.

It is your own responsibility to deal with these risks if you choose this path.

1.3 API Reference

`argon2_cffi` comes with an high-level API and hopefully reasonable defaults for Argon2 parameters that result in a verification time of between 0.5ms and 1ms on recent-ish hardware.

Unless you have any special needs, all you need to know is:

```
>>> from argon2 import PasswordHasher
>>> ph = PasswordHasher()
>>> hash = ph.hash("s3kr3tp4ssw0rd")
>>> hash
'$argon2i$v=19$m=512,t=2,p=2$5VtWOO3cGWYQHEMaYGbsfQ$AcmqasQgW/wI6wAHAMk4aQ'
>>> ph.verify(hash, "s3kr3tp4ssw0rd")
True
>>> ph.verify(hash, "t0t411ywr0ng")
Traceback (most recent call last):
  ...
argon2.exceptions.VerifyMismatchError: The password does not match the supplied hash
```

But of course the `PasswordHasher` class has all the parametrization you'll need:

class argon2.**PasswordHasher**(*time_cost=2*, *memory_cost=512*, *parallelism=2*, *hash_len=16*, *salt_len=16*, *encoding='utf-8'*)

High level class to hash passwords with sensible defaults.

Uses *always* Argon2**i** and a random salt.

The reason for this being a class is both for convenience to carry parameters and to verify the parameters only *once*. Any unnecessary slowdown when hashing is a tangible advantage for a brute force attacker.

Parameters

- **time_cost** (*int*) – Defines the amount of computation realized and therefore the execution time, given in number of iterations.
- **memory_cost** (*int*) – Defines the memory usage, given in kibibytes.
- **parallelism** (*int*) – Defines the number of parallel threads (*changes* the resulting hash value).
- **hash_len** (*int*) – Length of the hash in bytes.
- **salt_len** (*int*) – Length of random salt to be generated for each password.
- **encoding** (*str*) – The Argon2 C library expects bytes. So if `hash()` or `verify()` are passed an unicode string, it will be encoded using this encoding.

New in version 16.0.0.

hash(*password*)

Hash *password* and return an encoded hash.

Parameters **password** (`bytes` or `unicode`) – Password to hash.

Raises `argon2.exceptions.HashingError` – If hashing fails.

Return type unicode

verify(*hash*, *password*)

Verify that *password* matches *hash*.

Parameters

- **hash** (*unicode*) – An encoded hash as returned from `PasswordHasher.hash()`.
- **password** (`bytes` or `unicode`) – The password to verify.

Raises

- `argon2.exceptions.VerifyMismatchError` – If verification fails because *hash* is not valid for *secret* of *type*.
- `argon2.exceptions.VerificationError` – If verification fails for other reasons.

Returns `True` on success, raise `VerificationError` otherwise.

Return type bool

Changed in version 16.1.0: Raise `VerifyMismatchError` on mismatches instead of its more generic superclass.

If you don't specify any parameters, the following constants are used:

argon2.**DEFAULT_RANDOM_SALT_LENGTH**

argon2.**DEFAULT_HASH_LENGTH**

argon2.**DEFAULT_TIME_COST**

argon2.**DEFAULT_MEMORY_COST**

argon2.**DEFAULT_PARALLELISM**

You can see their values in `PasswordHasher`.

1.3.1 Exceptions

exception argon2.exceptions.**VerificationError**

Verification failed.

You can find the original error message from Argon2 in `args[0]`.

exception argon2.exceptions.**VerifyMismatchError**

The secret does not match the hash.

Subclass of `argon2.exceptions.VerificationError`.

New in version 16.1.0.

exception argon2.exceptions.**HashingError**

Raised if hashing failed.

You can find the original error message from Argon2 in `args[0]`.

Low Level

Low-level functions if you want to build your own higher level abstractions.

Warning: This is a "Hazardous Materials" module. You should **ONLY** use it if you're 100% absolutely sure that you know what you're doing because this module is full of land mines, dragons, and dinosaurs with laser guns.

class argon2.low_level.**Type**

Enum of Argon2 variants.

D = 0

Argon2**d** is faster and uses data-depending memory access, which makes it less suitable for hashing secrets and more suitable for cryptocurrencies and applications with no threats from side-channel timing attacks.

I = 1

Argon2**i** uses data-independent memory access, which is preferred for password hashing and password-based key derivation. Argon2i is slower as it makes more passes over the memory to protect from tradeoff attacks.

ID = 2

Argon2**id** is a hybrid of Argon2i and Argon2d, using a combination of data-depending and data-independent memory accesses, which gives some of Argon2i's resistance to side-channel cache timing attacks and much of Argon2d's resistance to GPU cracking attacks.

New in version 16.3.0.

argon2.low_level.**ARGON2_VERSION = 19**

The latest version of the Argon2 algorithm that is supported (and used by default).

New in version 16.1.0.

argon2.low_level.**hash_secret**(*secret*, *salt*, *time_cost*, *memory_cost*, *parallelism*, *hash_len*, *type*, *version=19*)

Hash *secret* and return an **encoded** hash.

An encoded hash can be directly passed into `verify_secret()` as it contains all parameters and the salt.

Parameters

- **secret** (*bytes*) – Secret to hash.
- **salt** (*bytes*) – A salt. Should be random and different for each secret.
- **type** (Type) – Which Argon2 variant to use.
- **version** (*int*) – Which Argon2 version to use.

For an explanation of the Argon2 parameters see `PasswordHasher`.

Return type bytes

Raises ***argon2.exceptions.HashingError*** – If hashing fails.

New in version 16.0.0.

```
>>> import argon2
>>> argon2.low_level.hash_secret(
...     b"secret", b"somesalt",
...     time_cost=1, memory_cost=8, parallelism=1, hash_len=64, type=argon2.low_level.
↪Type.D
```

(continues on next page)

(continued from previous page)

```
... )
b'$argon2d$v=19$m=8,t=1,p=1$c29tZXNhbHQ$ba2qC75j0+JAunZZ/
↪L0hZdQgCv+tOieBuKKXSrQiWm7nlkRcK+YqWr0i0m0WABJKelU8qHJp0SZzH0b1Z+ITvQ'
```

argon2.low_level.**verify_secret**(*hash*, *secret*, *type*)
Verify whether *secret* is correct for *hash* of *type*.

Parameters

- **hash** (*bytes*) – An encoded Argon2 hash as returned by *hash_secret()*.
- **secret** (*bytes*) – The secret to verify whether it matches the one in *hash*.
- **type** (Type) – Type for *hash*.

Raises

- ***argon2.exceptions.VerifyMismatchError*** – If verification fails because *hash* is not valid for *secret* of *type*.
- ***argon2.exceptions.VerificationError*** – If verification fails for other reasons.

Returns True on success, raise *VerificationError* otherwise.

Return type bool

New in version 16.0.0.

Changed in version 16.1.0: Raise *VerifyMismatchError* on mismatches instead of its more generic superclass.

The raw hash can also be computed:

argon2.low_level.**hash_secret_raw**(*secret*, *salt*, *time_cost*, *memory_cost*, *parallelism*, *hash_len*, *type*, *version=19*)
Hash *password* and return a **raw** hash.

This function takes the same parameters as *hash_secret()*.

New in version 16.0.0.

```
>>> argon2.low_level.hash_secret_raw(
...     b"secret", b"somesalt",
...     time_cost=1, memory_cost=8, parallelism=1, hash_len=8, type=argon2.low_level.
↪Type.D
... )
b'\xe4n\xf5\xc8|\xa3>\x1d'
```

The super low-level argon2_core() function is exposed too if you need access to very specific options:

argon2.low_level.**core**(*context*, *type*)
Direct binding to the argon2_ctx function.

Warning: This is a strictly advanced function working on raw C data structures. Both Argon2's and argon2_cffi's higher-level bindings do a lot of sanity checks and housekeeping work that *you* are now responsible for (e.g. clearing buffers). The structure of the *context* object can, has, and will change with *any* release!

Use at your own peril; argon2_cffi does *not* use this binding itself.

Parameters

- **context** – A CFFI Argon2 context object (i.e. an struct Argon2_Context/argon2_context).
- **type** (*int*) – Which Argon2 variant to use. You can use the `value` field of *Type*'s fields.

Return type int

Returns An Argon2 error code. Can be transformed into a string using `error_to_str()`.

New in version 16.0.0.

In order to use `core()`, you need access to `argon2_cffi`'s FFI objects. Therefore it is OK to use `argon2.low_level.ffi` and `argon2.low_level.lib` when working with it:

```
>>> from argon2.low_level import ARGON2_VERSION, Type, core, ffi, lib
>>> pwd = b"secret"
>>> salt = b"12345678"
>>> hash_len = 8
>>> # Make sure you keep FFI objects alive until *after* the core call!
>>> cout = ffi.new("uint8_t[]", hash_len)
>>> cpwd = ffi.new("uint8_t[]", pwd)
>>> csalt = ffi.new("uint8_t[]", salt)
>>> ctx = ffi.new(
...     "argon2_context *", dict(
...         version=ARGON2_VERSION,
...         out=cout, outlen=hash_len,
...         pwd=cpwd, pwdlen=len(pwd),
...         salt=csalt, saltlen=len(salt),
...         secret=ffi.NULL, secretlen=0,
...         ad=ffi.NULL, adlen=0,
...         t_cost=1,
...         m_cost=8,
...         lanes=1, threads=1,
...         allocate_cbk=ffi.NULL, free_cbk=ffi.NULL,
...         flags=lib.ARGON2_DEFAULT_FLAGS,
...     )
... )
>>> ctx
<cdata 'struct Argon2_Context *' owning 120 bytes>
>>> core(ctx, Type.D.value)
0
>>> out = bytes(ffi.buffer(ctx.out, ctx.outlen))
>>> out
b'\xb4\xe2HjO\x14d\x9b'
>>> out == argon2.low_level.hash_secret_raw(pwd, salt, 1, 8, 1, 8, Type.D)
True
```

All constants and types on `argon2.low_level.lib` are guaranteed to stay as long they are not altered by Argon2 itself.

`argon2.low_level.`**`error_to_str`**(*error*)

Convert an Argon2 error code into a native string.

Parameters **error** (*int*) – An Argon2 error code as returned by `core()`.

Return type str

New in version 16.0.0.

Deprecated APIs

These APIs are from the first release of `argon2_cffi` and proved to live in an unfortunate mid-level. On one hand they have defaults and check parameters but on the other hand they only consume byte strings.

Therefore the decision has been made to replace them by a high-level (*`argon2.PasswordHasher`*) and a low-level (*`argon2.low_level`*) solution. There are no immediate plans to remove them though.

`argon2.`**`hash_password`**`(`*password*, *salt=None*, *time_cost=2*, *memory_cost=512*, *parallelism=2*, *hash_len=16*, *type=<Type.I: 1>*`)`

Legacy alias for `hash_secret()` with default parameters.

Deprecated since version 16.0.0: Use *`argon2.PasswordHasher`* for passwords.

`argon2.`**`hash_password_raw`**`(`*password*, *salt=None*, *time_cost=2*, *memory_cost=512*, *parallelism=2*, *hash_len=16*, *type=<Type.I: 1>*`)`

Legacy alias for `hash_secret_raw()` with default parameters.

Deprecated since version 16.0.0: Use *`argon2.PasswordHasher`* for passwords.

`argon2.`**`verify_password`**`(`*hash*, *password*, *type=<Type.I: 1>*`)`

Legacy alias for `verify_secret()` with default parameters.

Deprecated since version 16.0.0: Use *`argon2.PasswordHasher`* for passwords.

1.4 Choosing Parameters

Note: You can probably just use *`argon2.PasswordHasher`* with its default values and be fine. Only tweak these if you've determined using *CLI* that these defaults are too slow or too fast for your use case.

Finding the right parameters for a password hashing algorithm is a daunting task. The authors of Argon2 specified a method in their paper but it should be noted that they also mention that no value for `time_cost` or `memory_cost` is actually insecure (cf. section 6.4).

1. Choose whether you want Argon2i or Argon2d (`type`). If you don't know what that means, choose Argon2i (`argon2.Type.I`).
2. Figure out how many threads can be used on each call to Argon2 (`parallelism`). They recommend twice as many as the number of cores dedicated to hashing passwords.
3. Figure out how much memory each call can afford (`memory_cost`).
4. Choose a salt length. 16 Bytes are fine.
5. Choose a hash length (`hash_len`). 16 Bytes are fine.
6. Figure out how long each call can take. One recommendation for concurent user logins is to keep it under 0.5ms.
7. Measure the time for hashing using your chosen parameters. Find a `time_cost` that is within your accounted time. If `time_cost=1` takes too long, lower `memory_cost`.

`argon2_cffi`'s *CLI* will help you with this process.

1.5 CLI

To aid you with finding the parameters, `argon2_cffi` offers a CLI interface that can be accessed using `python -m argon2`. It will benchmark Argon2's password *verification* in the current environment. You can use command line arguments to set hashing parameters:

```
$ python -m argon2 -t 1 -m 512 -p 2
Running Argon2i 100 times with:
hash_len: 16
memory_cost: 512
parallelism: 2
time_cost: 1

Measuring...

0.418ms per password verification
```

This should make it much easier to determine the right parameters for your use case and your environment.

1.6 Frequently Asked Questions

I'm using `bcrypt`/`scrypt`/`PBKDF2`, do I need to migrate? Using password hashes that aren't memory hard carries a certain risk but there's **no immediate danger or need for action**. If however you are deciding how to hash password *today*, pick Argon2 because it's a superior, future-proof choice.

But if you already use one of the hashes mentioned in the question, you should be fine for the foreseeable future.

Why do the `verify()` methods raise an Exception instead of returning `False`?

1. The Argon2 library had no concept of a "wrong password" error in the beginning. Therefore when writing these bindings, an exception with the full error had to be raised so you could inspect what went actually wrong.
2. In my opinion, a wrong password should raise an exception such that it can't pass unnoticed by accident. See also The Zen of Python: "Errors should never pass silently."
3. It's more Pythonic.

CHAPTER 2

Project Information

2.1 Backward Compatibility

`argon2_cffi` has a very strong backward compatibility policy. Generally speaking, you shouldn't ever be afraid of updating.

If breaking changes are needed do be done, they are:

1. . . . announced in the changelog.
2. . . . the old behavior raises a `DeprecationWarning` for a year.
3. . . . are done with another announcement in the changelog.

What explicitly *may* change over time are the default hashing parameters and the behavior of the *CLI*.

2.2 How To Contribute

Every open source project lives from the generous help by contributors that sacrifice their time and `argon2_cffi` is no different.

Here are a few guidelines to get you started:

- If you want to install a *development version* of `argon2_cffi` into your current `virtualenv`, you have to remember to:

  1. `git submodule init` (to initialize git submodule mechanics)
  2. `git submodule update` (to update the vendored Argon2 C library to the version `argon2_cffi` is currently packaging)
  3. `python setup.py build` (to build the CFFI module)
  4. `pip install -e .[dev]` (to [re-]install it along with the Python code and test dependencies)

  You have to perform steps 2, 3, and 4 whenever something changes in the Argon2 C code (e.g. if the vendored code has been updated).

- Try to limit each pull request to one change only.
- To run the test suite, all you need is a recent tox. It will ensure the test suite runs with all dependencies against all Python versions just as it will on Travis CI. If you lack some Python versions, you can can make it a non-failure using `tox --skip-missing-interpreters` (in that case you may want to look into pyenv that makes it very easy to install many different Python versions in parallel).

  One of the environments requires a system-wide installation of Argon2. On macOS, it's available in Homebrew and recent Ubuntus (zesty and later) ship it too.
- Make sure your changes pass our CI. You won't get any feedback until it's green unless you ask for it.
- Once you've addressed review feedback, make sure to bump the pull request with a short note, so we know you're done.
- If your change is noteworthy, add an entry to the changelog. Use semantic newlines and add a link to your pull request.
- No contribution is too small; please submit as many fixes for typos and grammar bloopers as you can!
- Don't break backward compatibility.
- *Always* add tests and docs for your code. This is a hard rule; patches with missing tests or documentation won't be merged.
- Write good test docstrings.
- Obey PEP 8 and PEP 257.

Please note that this project is released with a Contributor Code of Conduct. By participating in this project you agree to abide by its terms. Please report any harm to Hynek Schlawack in any way you find appropriate.

Thank you for considering to contribute!

2.3 Changelog

Versions are year-based with a strict backward compatibility policy. The third digit is only for regressions.

2.3.1 18.1.0 (2018-01-06)

Vendoring Argon2 @ 670229c (20171227)

Changes:

- It is now possible to use the `argon2_cffi` bindings against an Argon2 library that is provided by the system.

2.3.2 16.3.0 (2016-11-10)

Vendoring Argon2 @ 1c4fc41f81f358283755eea88d4ecd05e43b7fd3 (20161029)

Changes:

- Prevent side-effects like the installation of `cffi` if `setup.py` is called with a command that doesn't require it. #20
- Fix a bunch of warnings with new `cffi` versions and Python 3.6. #14 #16
- Add low-level bindings for Argon2id functions.

2.3.3 16.2.0 (2016-09-10)

Vendoring Argon2 @ 4844d2fee15d44cb19296ddf36029326d17c5aa3

Changes:

- Fix compilation on debian jessie. #13

2.3.4 16.1.0 (2016-04-19)

Vendoring Argon2 @ 00aaa6604501fade85853a4b2f5695611ff6e7c5.

Backward-incompatible changes:

- Python 3.3 and 2.6 aren't supported anymore. They may work by chance but any support to them has been ceased.

  The last Python 2.6 release was on October 29, 2013 and isn't supported by the CPython core team anymore. Major Python packages like Django and Twisted dropped Python 2.6 a while ago already.

  Python 3.3 never had a significant user base and wasn't part of any distribution's LTS release.

Changes:

- Add `VerifyMismatchError` that is raised if verification fails only because of a password/hash mismatch. It's a subclass of `VerificationError` therefore this change is completely backward compatible.
- Add support for Argon2 1.3. Old hashes remain functional but opportunistic rehashing is strongly recommended.

2.3.5 16.0.0 (2016-01-02)

Vendoring Argon2 @ 421dafd2a8af5cbb215e16da5953663eb101d139.

Deprecations:

- `hash_password()`, `hash_password_raw()`, and `verify_password()` should not be used anymore. For hashing passwords, use the new `argon2.PasswordHasher`. If you want to implement your own higher-level abstractions, use the new low-level APIs `hash_secret()`, `hash_secret_raw()`, and `verify_secret()` from the `argon2.low_level` module. If you want to go *really* low-level, `core()` is for you. The old functions will *not* raise any warnings though and there are *no* immediate plans to remove them.

Changes:

- Add `argon2.PasswordHasher`. A higher-level class specifically for hashing passwords that also works on Unicode strings.
- Add `argon2.low_level` module with low-level API bindings for building own high-level abstractions.

2.3.6 15.0.1 (2015-12-18)

Vendoring Argon2 @ 4fe0d8cda37691228dd5a96a310be57369403a4b.

Changes:

- Fix `long_description` on PyPI.

2.3.7 15.0.0 (2015-12-18)

Vendoring Argon2 @ 4fe0d8cda37691228dd5a96a310be57369403a4b.

Changes:

- `verify_password()` doesn't guess the hash type if passed `None` anymore. Supporting this resulted in measurable overhead (~ 0.6ms vs 0.8ms on my notebook) since it had to happen in Python. That means that naïve usage of the API would give attackers an edge. The new behavior is that it has the same default value as `hash_password()` such that `verify_password(hash_password(b"password"), b"password")` still works.
- Conditionally use the SSE2-optimized version of `argon2` on x86 architectures.
- More packaging fixes. Most notably compilation on Visual Studio 2010 for Python 3.3 and 3.4.
- Tweaked default parameters to more reasonable values. Verification should take between 0.5ms and 1ms on recent-ish hardware.

2.3.8 15.0.0b5 (2015-12-10)

Vendoring Argon2 @ 4fe0d8cda37691228dd5a96a310be57369403a4b.

Initial work. Previous betas were only for fixing Windows packaging. The authors of Argon2 were kind enough to help me to get it building under Visual Studio 2008 that we're forced to use for Python 2.7 on Windows.

2.4 Credits & License

`argon2_cffi` is maintained by Hynek Schlawack and released under the MIT license.

The development is kindly supported by Variomedia AG.

A full list of contributors can be found in GitHub's overview.

2.4.1 Vendored Code

Argon2

The original Argon2 repo can be found at https://github.com/P-H-C/phc-winner-argon2/.

Except for the components listed below, the Argon2 code in this repository is copyright (c) 2015 Daniel Dinu, Dmitry Khovratovich (main authors), Jean-Philippe Aumasson and Samuel Neves, and under CC0 license.

The string encoding routines in src/encoding.c are copyright (c) 2015 Thomas Pornin, and under CC0 license.

The BLAKE2 code in `src/blake2/` is copyright (c) Samuel Neves, 2013-2015, and under CC0 license.

The authors of Argon2 also were very helpful to get the library to compile on ancient versions of Visual Studio for ancient versions of Python.

The documentation also quotes frequently from the Argon2 paper to avoid mistakes by rephrasing.

msinttypes

In order to be able to compile on Visual Studio 2008 and Visual Studio 2010 which are required for Python 2.7 and 3.4 respectively, we also ship two C headers with integer types. They are from the msinttypes project (auto-import on GitHub) and licensed under New BSD:

CHAPTER 3

Indices and tables

- genindex
- search

Python Module Index

Index