
argon2*cf fi*Documentation

Release 19.1.0

Hynek Schlawack

Aug 19, 2022

CONTENTS

1	User's Guide	3
2	Project Information	11
3	Indices and tables	21
	Python Module Index	23
	Index	25

Release v19.1.0 (*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
'$argon2id$v=19$m=102400,t=2,p=8$tSm+JOWigOgPZx/g44K5fQ$WDyus6py50bVFIPkja281Q'
>>> ph.verify(hash, "s3kr3tp4ssw0rd")
True
>>> ph.check_needs_rehash(hash)
False
>>> ph.verify(hash, "t0t41lywr0ng")
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.

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-dependent 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-dependent 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

```
python -m 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:

```
python -m 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 preferred 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 \  
python -m pip install --no-binary=argon2_cffi argon2_cffi
```

This approach can lead to problems around your build chain and you can run into incompatibilities 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 40–50ms on recent-ish hardware.

Warning: The current memory requirement is set to rather conservative 100 MB. However, in memory constrained environments like Docker containers that can lead to problems. One possible non-obvious symptom are apparent freezes that are caused by swapping.

Please check *Choosing Parameters* for more details.

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

```
>>> from argon2 import PasswordHasher
>>> ph = PasswordHasher()
>>> hash = ph.hash("s3kr3tp4ssw0rd")
>>> hash
'$argon2id$v=19$m=102400,t=2,p=8$tSm+JOWigOgPZx/g44K5fQ$WDyus6py50bVFIPkjA28lQ'
>>> ph.verify(hash, "s3kr3tp4ssw0rd")
True
>>> ph.check_needs_rehash(hash)
False
>>> ph.verify(hash, "t0t411ywr0ng")
Traceback (most recent call last):
...
argon2.exceptions.VerifyMismatchError: The password does not match the supplied hash
```

A login function could thus look like this:

```
import argon2

ph = argon2.PasswordHasher()

def login(db, user, password):
    hash = db.get_password_hash_for_user(user)

    # Verify password, raises exception if wrong.
    ph.verify(hash, password)

    # Now that we have the cleartext password,
    # check the hash's parameters and if outdated,
    # rehash the user's password in the database.
    if ph.check_needs_rehash(hash):
        db.set_password_hash_for_user(user, ph.hash(password))
```

While the PasswordHasher class has the aspiration to be good to use out of the box, it has all the parametrization you'll need:

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]`.

exception argon2.exceptions.InvalidHash

Raised if the hash is invalid before passing it to Argon2.

New in version 18.2.0.

1.3.2 Utilities

1.3.3 Low Level

```
>>> 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
... )
b'$argon2d$v=19$m=8,t=1,p=1$c29tZXNhbHQ$ba2qC75j0+JAunZZ/
↪L0hZdQgCv+t0ieBuKKXSrQiWm7nlkRcK+YqWr0i0m0WABJKelU8qHJp0SZZH0b1Z+ITvQ'
```

The raw hash can also be computed:

```
>>> 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
```

(continues on next page)

(continued from previous page)

```
... )
b'\xe4n\xf5\xc8|\xa3>\x1d'
```

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

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, pwrlen=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\xe2Hj0\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.

1.3.4 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.

1.4 Choosing Parameters

Note: You can probably just use `argon2.PasswordHasher` with its default values and be fine. But it's good to double check using `argon2_cffi`'s *CLI* client, whether its 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](#), however some parts of it have been revised in the [RFC draft](#) for Argon2 that is currently being written.

The current recommended best practice is as follow:

1. Choose whether you want Argon2i, Argon2d, or Argon2id (type). If you don't know what that means, choose Argon2id (`argon2.Type.ID`).
2. Figure out how many threads can be used on each call to Argon2 (parallelism, called "lanes" in the RFC). They recommend twice as many as the number of cores dedicated to hashing passwords. `PasswordHasher` will *not* determine this for you and use a default value that you can find in the linked API docs.
3. Figure out how much memory each call can afford (`memory_cost`). The RFC recommends 4 GB for backend authentication and 1 GB for frontend authentication. The APIs use [Kibibytes](#) (1024 bytes) as base unit.
4. Select the salt length. 16 bytes is sufficient for all applications, but can be reduced to 8 bytes in the case of space constraints.
5. Choose a hash length (`hash_len`, called "tag length" in the documentation). 16 bytes is sufficient for password verification.
6. Figure out how long each call can take. One [recommendation](#) for concurrent user logins is to keep it under 0.5 ms. The RFC recommends under 500 ms. The truth is somewhere between those two values: more is more secure, less is a better user experience. `argon2_cffi`'s defaults try to land somewhere in the middle and aim for ~50ms, but the actual time depends on your hardware.

Please note though, that even a verification time of 1 second won't protect you against bad passwords from the "top 10,000 passwords" lists that you can find online.

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
Running Argon2id 100 times with:
hash_len: 16 bytes
memory_cost: 102400 KiB
parallelism: 8 threads
time_cost: 2 iterations

Measuring...

45.3ms 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`/`PBKDF2`/`scrypt`/`yescrypt`, 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*, Argon2 is the 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. If you're using `scrypt` or `yescrypt`, you will be probably fine for good.

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.
It goes without saying that it's impossible to switch now for backward-compatibility reasons.
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*.

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 to 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

First off, thank you for considering contributing to `argon2_cffi`! It's people like *you* who make it such a great tool for everyone.

This document intends to make contribution more accessible by codifying tribal knowledge and expectations. Don't be afraid to open half-finished PRs, and ask questions if something is unclear!

2.2.1 Workflow

- No contribution is too small! Please submit as many fixes for typos and grammar bloopers as you can!
- Try to limit each pull request to *one* change only.
- Since we squash on merge, it's up to you how you handle updates to the master branch. Whether you prefer to rebase on master or merge master into your branch, do whatever is more comfortable for you.
- *Always* add tests and docs for your code. This is a hard rule; patches with missing tests or documentation can't be merged.
- 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.
- Don't break `backward compatibility`.

2.2.2 Code

- Obey [PEP 8](#) and [PEP 257](#). We use the `"""`-on-separate-lines style for docstrings:

```
def func(x):  
    """  
    Do something.  
  
    :param str x: A very important parameter.  
  
    :rtype: str  
    """
```

- If you add or change public APIs, tag the docstring using `.. versionadded:: 16.0.0 WHAT` or `.. versionchanged:: 16.2.0 WHAT`.
- We use `isort` to sort our imports, and we follow the [Black](#) code style with a line length of 79 characters. As long as you run our full tox suite before committing, or install our [pre-commit](#) hooks (ideally you'll do both – see below “Local Development Environment”), you won't have to spend any time on formatting your code at all. If you don't, CI will catch it for you – but that seems like a waste of your time!

2.2.3 Tests

- Write your asserts as `expected == actual` to line them up nicely:

```
x = f()  
  
assert 42 == x.some_attribute  
assert "foo" == x._a_private_attribute
```

- 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 always limit the environments like `tox -e py27,py35` (in that case you may want to look into [pyenv](#), which makes it very easy to install many different Python versions in parallel).
- Write [good test docstrings](#).

2.2.4 Documentation

- Use [semantic newlines](#) in `reStructuredText` files (files ending in `.rst`):

```
This is a sentence.  
This is another sentence.
```

- If you start a new section, add two blank lines before and one blank line after the header, except if two headers follow immediately after each other:

```
Last line of previous section.
```

```
Header of New Top Section  
-----
```

(continues on next page)

(continued from previous page)

Header of New Section

AAAAAAAAAAAAAAAAAAAAAAAAAAAA

First line of new section.

- If your change is noteworthy, add an entry to the changelog. Use [semantic newlines](#), and add a link to your pull request:

```
- Added ``argon2_cffi.func()`` that does foo.
  It's pretty cool.
  [#1 <https://github.com/hynek/argon2_cffi/pull/1>`_]
- ``argon2_cffi.func()`` now doesn't crash the Large Hadron Collider anymore.
  That was a nasty bug!
  [#2 <https://github.com/hynek/argon2_cffi/pull/2>`_]`_
```

2.2.5 Local Development Environment

You can (and should) run our test suite using `tox`. However, you'll probably want a more traditional environment as well. We highly recommend to develop using the latest Python 3 release because `argon2_cffi` tries to take advantage of modern features whenever possible.

First create a [virtual environment](#). It's out of scope for this document to list all the ways to manage virtual environments in Python, but if you don't already have a pet way, take some time to look at tools like `pew`, `virtualfish`, and `virtualenvwrapper`.

Next, get an up to date checkout of the `argon2_cffi` repository:

```
$ git clone git@github.com:hynek/argon2_cffi.git
```

or if you want to use git via https:

```
$ git clone https://github.com/hynek/argon2_cffi.git
```

Change into the newly created directory and **after activating your virtual environment** install an editable version of `argon2_cffi` along with its tests and docs requirements:

- First you have to make sure, that our git submodules are up to date and the Argon2 extension is built:
 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)

One of the environments requires a system-wide installation of Argon2. On macOS, it's available in Homebrew (`brew install argon2`, but you also will have to update your `LD_FLAGS` so your compiler finds it) and recent Ubuntu (zesty and later) ship it too.

- Next (re-)install `argon2_cffi` along with its development requirements:

```
$ pip install -e '.[dev]'
```

Whenever the Argon2 C code changes: you will have to perform the steps above again except of `git submodule init`.

At this point,

```
$ python -m pytest
```

should work and pass, as should:

```
$ cd docs
$ make html
```

The built documentation can then be found in docs/_build/html/.

To avoid committing code that violates our style guide, we strongly advise you to install [pre-commit](#)¹ hooks:

```
$ pre-commit install
```

You can also run them anytime (as our tox does) using:

```
$ pre-commit run --all-files
```

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 19.1.0 (2019-01-17)

Vendoring Argon2 @ 670229c (20171227)

Backward-incompatible changes:

none

Deprecations:

none

¹ pre-commit should have been installed into your virtualenv automatically when you ran `pip install -e '[dev]'` above. If pre-commit is missing, it may be that you need to re-run `pip install -e '[dev]'`.

Changes:

- Added support for Argon2 v1.2 hashes in `argon2.extract_parameters()`.
-

2.3.2 18.3.0 (2018-08-19)

Vendoring Argon2 @ 670229c (20171227)

Backward-incompatible changes:

none

Deprecations:

none

Changes:

- `argon2.PasswordHasher`'s hash type is configurable now.
-

2.3.3 18.2.0 (2018-08-19)

Vendoring Argon2 @ 670229c (20171227)

Changes:

- The hash type for `argon2.PasswordHasher` is `Argon2id` now.
This decision has been made based on the recommendations in the latest [Argon2 RFC draft](#). #33 #34
 - To make the change of hash type backward compatible, `argon2.PasswordHasher.verify()` now determines the type of the hash and verifies it accordingly.
 - Some of the hash parameters have been made stricter to be closer to said recommendations. The current goal for a hash verification times is around 50ms. #41
 - To allow for bespoke decisions about upgrading Argon2 parameters, it's now possible to extract them from a hash via the `argon2.extract_parameters()` function. #41
 - Additionally `argon2.PasswordHasher` now has a `check_needs_rehash()` method that allows to verify whether a hash has been created with the instance's parameters or whether it should be rehashed. #41
-

2.3.4 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.
-

2.3.5 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.
-

2.3.6 16.2.0 (2016-09-10)

Vendoring Argon2 @ 4844d2fee15d44cb19296ddf36029326d17c5aa3

Changes:

- Fix compilation on debian jessie. #13
-

2.3.7 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.
-

2.3.8 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.
-

2.3.9 15.0.1 (2015-12-18)

Vendoring Argon2 @ 4fe0d8cda37691228dd5a96a310be57369403a4b.

Changes:

- Fix `long_description` on PyPI.
-

2.3.10 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.
-

2.3.11 15.0.0b5 (2015-12-10)

Vendoring Argon2 @ [4fe0d8cda37691228dd5a96a310be57369403a4b](https://github.com/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:

Copyright (c) 2006-2013 Alexander Chemeris

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. Neither the name of the product nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

INDICES AND TABLES

- genindex
- search

PYTHON MODULE INDEX

a

argon2, 5

INDEX

A

argon2
 module, 5

D

DEFAULT_HASH_LENGTH (*in module argon2*), 6
DEFAULT_MEMORY_COST (*in module argon2*), 6
DEFAULT_PARALLELISM (*in module argon2*), 6
DEFAULT_RANDOM_SALT_LENGTH (*in module argon2*), 5
DEFAULT_TIME_COST (*in module argon2*), 6

H

HashingError, 6

I

InvalidHash, 6

M

module
 argon2, 5

V

VerificationError, 6
VerifyMismatchError, 6