argon2-cffi Documentation

Release 21.3.0

Hynek Schlawack

Aug 19, 2022

CONTENTS

Release v21.3.0 (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("correct horse battery staple")
>>> hash
'$argon2id$v=19$m=65536,t=3,p=4$MIIRqgvgQbgj220jfp0MPA$YfwJSVjtjSU0zzV/P3S9nnQ/
↪USre2wvJMjfCIjrTQbg'
>>> ph.verify(hash, "correct horse battery staple")
True
>>> ph.check_needs_rehash(hash)
False
>>> ph.verify(hash, "Tr0ub4dor&3")
Traceback (most recent call last):
  ...
argon2.exceptions.VerifyMismatchError: The password does not match the supplied hash
```

CHAPTER ONE

USER'S GUIDE

1.1 What is *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 a Vendored *Argon2*

```
python -m pip install argon2-cffi
```

should be all it takes.

But since *argon2-cffi* depends on argon2-cffi-bindings that 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* packages 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 by argon2-cffi-bindings. With a recent-enough *pip* and *setuptools*, they should be used automatically.

Source Distribution

A working C compiler and CFFI environment are required to build the argon2-cffi-bindings dependency. 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-bindings* 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 of argon2-cffi-bindings:

```
env ARGON2_CFFI_USE_SYSTEM=1 \
  python -m pip install --no-binary=argon2-cffi-bindings 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.

Available since version 18.1.0. The `--no-binary` option value changed in 21.2.0 due to the outsourcing of the binary bindings.

1.2.3 Override Automatic SSE2 Detection

Usually the build process tries to guess whether or not it should use SSE2-optimized code. Despite our best efforts, this can go wrong.

Therefore you can use the `ARGON2_CFFI_USE_SSE2` environment variable to control the process:

- If you set it to `1`, *argon2-cffi* will build **with** SSE2 support.
- If you set it to `0`, *argon2-cffi* will build **without** SSE2 support.
- If you set it to anything else, it will be ignored and *argon2-cffi* will try to guess.

Available since version 20.1.0.

1.3 API Reference

argon2-cffi comes with an high-level API and uses the officially recommended low-memory *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 64 MB. However, in memory constrained environments such as *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("correct horse battery staple")
>>> hash
'$argon2id$v=19$m=65536,t=3,p=4$MIIRqgvgQbgj220jfp0MPA$YfwJSVjtjSU0zzV/P3S9nnQ/
↪USre2wvJMjfCIjrTQbg'
>>> ph.verify(hash, "correct horse battery staple")
True
>>> ph.check_needs_rehash(hash)
False
>>> ph.verify(hash, "Tr0ub4dor&3")
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()

```

(continues on next page)

(continued from previous page)

```
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:

class argon2.**PasswordHasher**(*time_cost: int = 3*, *memory_cost: int = 65536*, *parallelism: int = 4*, *hash_len: int = 32*, *salt_len: int = 16*, *encoding: str = 'utf-8'*, *type: Type = Type.ID*)

High level class to hash passwords with sensible defaults.

Uses Argon2**id** by default and always uses a random salt for hashing. But it can verify any type of *Argon2* as long as the hash is correctly encoded.

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 a `str`, it will be encoded using this encoding.
- **type** (Type) – *Argon2* type to use. Only change for interoperability with legacy systems.

New in version 16.0.0.

Changed in version 18.2.0: Switch from Argon2i to Argon2id based on the recommendation by the current RFC draft. See also *Choosing Parameters*.

Changed in version 18.2.0: Changed default *memory_cost* to 100 MiB and default *parallelism* to 8.

Changed in version 18.2.0: `verify` now will determine the type of hash.

Changed in version 18.3.0: The *Argon2* type is configurable now.

New in version 21.2.0: `from_parameters()`

Changed in version 21.2.0: Changed defaults to `argon2.profiles.RFC_9106_LOW_MEMORY`.

check_needs_rehash(*hash: str*) → bool

Check whether *hash* was created using the instance's parameters.

Whenever your *Argon2* parameters – or *argon2-cffi*'s defaults! – change, you should rehash your passwords at the next opportunity. The common approach is to do that whenever a user logs in, since that should be the only time when you have access to the cleartext password.

Therefore it's best practice to check – and if necessary rehash – passwords after each successful authentication.

Return type
bool

New in version 18.2.0.

classmethod from_parameters(*params:* Parameters) → *PasswordHasher*

Construct a *PasswordHasher* from *params*.

New in version 21.2.0.

hash(*password: Union[str, bytes]*) → str

Hash *password* and return an encoded hash.

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

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

Return type
str

verify(*hash: Union[str, bytes], password: Union[str, bytes]*) → Literal[True]

Verify that *password* matches *hash*.

> **Warning:** It is assumed that the caller is in full control of the hash. No other parsing than the determination of the hash type is done by *argon2-cffi*.

Parameters

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

Raises

- `argon2.exceptions.VerifyMismatchError` – If verification fails because *hash* is not valid for *password*.
- `argon2.exceptions.VerificationError` – If verification fails for other reasons.
- `argon2.exceptions.InvalidHash` – If *hash* is so clearly invalid, that it couldn't be passed to *Argon2*.

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.

New in version 18.2.0: Hash type agility.

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**

They are taken from *argon2.profiles.RFC_9106_LOW_MEMORY*.

1.3.1 Profiles

This module offers access to standardized parameters that you can load using `PasswordHasher.from_parameters()`. See the source code for concrete values and *Choosing Parameters* for more information.

New in version 21.2.0.

You can try them out using the *CLI* interface. For example:

```
$ python -m argon2 --profile RFC_9106_HIGH_MEMORY
Running Argon2id 100 times with:
hash_len: 32 bytes
memory_cost: 2097152 KiB
parallelism: 4 threads
time_cost: 1 iterations

Measuring...

866.5ms per password verification
```

That should give you a feeling on how they perform in *your* environment.

argon2.profiles.**RFC_9106_HIGH_MEMORY**

Called "FIRST RECOMMENDED option" by RFC 9106.

Requires beefy 2 GiB, so be careful in memory-contrained systems.

New in version 21.2.0.

argon2.profiles.**RFC_9106_LOW_MEMORY**

Called "SECOND RECOMMENDED option" by RFC 9106.

The main difference is that it only takes 64 MiB of RAM.

The values from this profile are the default parameters used by *argon2.PasswordHasher*.

New in version 21.2.0.

`argon2.profiles.PRE_21_2`

The default values that *argon2-cffi* used from 18.2.0 until 21.2.0.

Needs 100 MiB of RAM.

New in version 21.2.0.

`argon2.profiles.CHEAPEST`

This is the cheapest-possible profile.

> **Warning:** This is only for testing purposes! Do **not** use in production!

New in version 21.2.0.

1.3.2 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.

1.3.3 Utilities

`argon2.extract_parameters`(*hash: str*) → *Parameters*

Extract parameters from an encoded *hash*.

Parameters
`params` (*str*) – An encoded Argon2 hash string.

Return type
Parameters

New in version 18.2.0.

class `argon2.Parameters`(*type: Type, version: int, salt_len: int, hash_len: int, time_cost: int, memory_cost: int, parallelism: int*)

Argon2 hash parameters.

See *Choosing Parameters* on how to pick them.

Variables

- **type** (Type) – Hash type.
- **version** (*int*) – Argon2 version.
- **salt_len** (*int*) – Length of the salt in bytes.
- **hash_len** (*int*) – Length of the hash in bytes.
- **time_cost** (*int*) – Time cost in iterations.
- **memory_cost** (*int*) – Memory cost in kibibytes.
- **parallelism** (*int*) – Number of parallel threads.

New in version 18.2.0.

1.3.4 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(value)`

Enum of Argon2 variants.

Please see *Choosing Parameters* on how to pick one.

`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. 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.

That makes it the preferred type for password hashing and password-based key derivation.

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: bytes, salt: bytes, time_cost: int, memory_cost: int, parallelism: int, hash_len: int, type: Type, version: int = 19) → bytes`

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

`argon2.low_level.`**`verify_secret`**(*hash: bytes*, *secret: bytes*, *type: Type*) → Literal[True]

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: bytes*, *salt: bytes*, *time_cost: int*, *memory_cost: int*, *parallelism: int*, *hash_len: int*, *type: Type*, *version: int = 19*) → bytes

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: Any*, *type: int*) → int

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,
```

(continues on next page)

(continued from previous page)

```
...         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: int*`)` → str

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.

1.3.5 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: bytes, salt: Optional[bytes] = None, time_cost: int = 3, memory_cost: int = 65536, parallelism: int = 4, hash_len: int = 32, type: Type = Type.I*`)` → bytes

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

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

`argon2.`**`hash_password_raw`**`(`*password: bytes, salt: Optional[bytes] = None, time_cost: int = 3, memory_cost: int = 65536, parallelism: int = 4, hash_len: int = 32, type: Type = Type.I*`)` → bytes

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

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

`argon2.`**`verify_password`**`(`*hash: bytes, password: bytes, type: Type = Type.I*`)` → Literal[True]

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. 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. As of September 2021, we have the official Internet standard RFC 9106 to help use with it.

It comes with two recommendations in section 4, that (as of *argon2-cffi* 21.2.0) you can load directly from the `argon2.profiles` module: `argon2.profiles.RFC_9106_HIGH_MEMORY` (called "FIRST RECOMMENDED") and `argon2.profiles.RFC_9106_LOW_MEMORY` ("SECOND RECOMMENDED") into `argon2.PasswordHasher.from_parameters()`.

Please use the *CLI* interface together with its *--profile* argument to see if they work for you.

If you need finer tuning, 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 4 threads.
3. Figure out how much memory each call can afford (`memory_cost`). 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 used to recommend under 500 ms. The truth is somewhere between those two values: more is more secure, less is a better user experience. *argon2-cffi*'s current defaults land with ~50ms somewhere in the middle, 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. Start with `time_cost=1` and measure the time it takes. Raise `time_cost` until it 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.

Note: Alternatively, you can also refer to the OWASP cheatsheet.

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:

```
$ python -m argon2
Running Argon2id 100 times with:
hash_len: 32 bytes
memory_cost: 65536 KiB
parallelism: 4 threads
time_cost: 3 iterations

Measuring...

45.7ms per password verification
```

You can use command line arguments to set hashing parameters. Either by setting them one by one (`-t` for time, `-m` for memory, `-p` for parallelism, `-l` for hash length), or by passing `--profile` followed by one of the names from *`argon2.profiles`*. In that case, the other parameters are ignored. If you don't pass any arguments as above, it runs with *`argon2.PasswordHasher`*'s default values.

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.

CHAPTER TWO

PROJECT INFORMATION

argon2-cffi is available from PyPI, the documentation lives at Read the Docs, the code on GitHub. The low-level Argon2 CFFI bindings are maintained in the separate project argon2-cffi-bindings.

It targets Python 3.6 and newer, and PyPy3. The last version that works with Python 2.7 is 20.1.0, and the last version that works with Python 3.5 is 21.1.0.

argon2-cffi implements *Argon2* version 1.3, as described in Argon2: the memory-hard function for password hashing and other applications.

2.1 *argon2-cffi* for Enterprise

Available as part of the Tidelift Subscription.

The maintainers of *argon2-cffi* and thousands of other packages are working with Tidelift to deliver commercial support and maintenance for the open source packages you use to build your applications. Save time, reduce risk, and improve code health, while paying the maintainers of the exact packages you use. Learn more.

2.1.1 Credits & License

argon2-cffi is maintained by Hynek Schlawack and released under the MIT license.

The development is kindly supported by Variomedia AG. Please consider supporting me too!

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

CHAPTER THREE

INDICES AND TABLES

- genindex
- search

PYTHON MODULE INDEX

a

INDEX