yespower spec website: https://www.openwall.com/yespower/

What is yespower?

yespower is a proof-of-work (PoW) focused fork of yescrypt. While yescrypt is a password-based key derivation function (KDF) and password hashing scheme, and thus is meant for processing passwords, yespower is meant for processing trial inputs such as block headers (including nonces) in PoW-based blockchains.

On its own, yespower isn't a complete proof-of-work system. Rather, in the blockchain use case, yespower's return value is meant to be checked for being numerically no greater than the blockchain's current target (which is related to mining difficulty) or else the proof attempt (yespower invocation) is to be repeated (with a different nonce) until the condition is finally met (allowing a new block to be mined). This process isn't specific to yespower and isn't part of yespower itself (rather, it is similar in many PoW-based blockchains and is to be defined and implemented externally to yespower) and thus isn't described in here any further.

Why or why not yespower?

Different proof-of-work schemes in existence vary in many aspects, including in friendliness to different types of hardware. There's demand for all sorts of hardware (un)friendliness in those - for different use cases and by different communities.

yespower in particular is designed to be CPU-friendly, GPU-unfriendly, and FPGA/ASIC-neutral. In other words, it's meant to be relatively efficient to compute on current CPUs and relatively inefficient on current GPUs. Unfortunately, being GPU-unfriendly also means that eventual FPGA and ASIC implementations will only compete with CPUs, and at least ASICs will win over the CPUs (FPGAs might not because of this market's peculiarities - large FPGAs are even more "over-priced" than large CPUs are), albeit by far not to the extent they did e.g. for Bitcoin and Litecoin.

There's a lot of talk about "ASIC resistance". What is (or should be) meant by that is limiting the advantage of specialized ASICs. While limiting the advantage at KDF to e.g. 10x and at password hashing to e.g. 100x (talking orders of magnitude here, in whatever terms) may be considered "ASIC resistant" (as compared to e.g. 100,000x we'd have without trying), similar improvement factors are practically not "ASIC resistant" for cryptocurrency mining where they can make all the difference between CPU mining being profitable and not. There might also exist in-between PoW use cases where moderate ASIC advantage is OK, such as with non-cryptocurrency and/or private/permissioned blockchains.

Thus, current yespower may be considered either a short-term choice (valid until one of its uses provides sufficient perceived incentive to likely result in specialized ASICs) or a deliberate choice of a pro-CPU, anti-GPU, moderately-pro-ASIC PoW scheme. It is also possible to respond to known improvements in future GPUs/implementations and/or to ASICs with new versions of yespower that users would need to switch to.

yespower versions.

yespower includes optimized and specialized re-implementation of the obsolete yescrypt 0.5 (based off its first submission to Password Hashing Competition back in 2014) now re-released as yespower 0.5, and brand new proof-of-work specific variation known as yespower 1.0.

yespower 0.5 is intended as a compatible upgrade for cryptocurrencies that already use yescrypt 0.5 (providing a few percent speedup), and yespower 1.0 may be used as a further upgrade or a new choice of PoW by those and other cryptocurrencies and other projects.

There are many significant differences between yespower 0.5 and 1.0 under the hood, but the main user visible difference is yespower 1.0 greatly improving on GPU-unfriendliness in light of improvements seen in modern GPUs (up to and including NVIDIA Volta) and GPU implementations of yescrypt 0.5. This is achieved mostly through greater use of CPUs' L2 cache.

The version of algorithm to use is requested through parameters, allowing for both algorithms to co-exist in client and miner implementations (such as in preparation for a cryptocurrency hard-fork and/or supporting multiple cryptocurrencies in one program).

Parameter selection.

For new uses of yespower, set the requested version to the highest supported, and set N*r to the highest you can reasonably afford in terms of proof verification time (which might in turn be determined by desired share rate per mining pool server), using one of the following options:

1 MiB: N = 1024, r = 8
2 MiB: N = 2048, r = 8
4 MiB: N = 1024, r = 32
8 MiB: N = 2048, r = 32
16 MiB: N = 4096, r = 32

and so on for higher N keeping r=32.

You may also set the personalization string to your liking, but that is not required (you can set its pointer to NULL and its length to 0). Its support is provided mostly for compatibility with existing modifications of yescrypt 0.5.

Performance.

Please refer to PERFORMANCE for some benchmarks and performance tuning.

How to test yespower for proper operation.

On a Unix-like system, invoke "make check". This will build and run a program called "tests", and check its output against the supplied file TESTS-OK. If everything matches, the final line of output should be the word "PASSED".

We do most of our testing on Linux systems with gcc. The supplied Makefile assumes that you use gcc.

Alternate code versions and make targets.

Two implementations of yespower are included: reference and optimized. By default, the optimized implementation is built. Internally, the optimized implementation uses conditional compilation to choose between usage of various SIMD instruction sets where supported and scalar code.

The reference implementation is unoptimized and is very slow, but it has simpler and shorter source code. Its purpose is to provide a simple human- and machine-readable specification that implementations intended for actual use should be tested against. It is deliberately mostly not optimized, and it is not meant to be used in production.

Similarly to "make check", there's "make check-ref" to build and test the reference implementation. There's also "make ref" to build the reference implementation and have the "benchmark" program use it.

"make clean" may need to be run between making different builds.

How to integrate yespower in a program.

Although yespower.h provides several functions, chances are that you will only need to use yespower_tls(). Please see the comment on this function in yespower.h and its example usage in tests.c and benchmark.c, including parameter sets requesting yescrypt 0.5 as used by certain existing cryptocurrencies.

To integrate yespower in an altcoin based on Bitcoin Core, you might invoke yespower_tls() from either a maybe-new (depending on where you fork from) CBlockHeader::GetPoWHash() (and invoke that where PoW is needed like e.g. Litecoin does for scrypt) or CBlockHeader::GetHash() (and implement caching for its return value like e.g. YACoin does for scrypt). Further detail on this (generating new genesis blocks, etc.) is not yespower-specific and thus is not provided here. Just like (and even more so than) yespower itself, this guidance is provided as-is and without guarantee of being correct and safe to follow. You're supposed to know what you're doing.

Credits.

scrypt has been designed by Colin Percival. yescrypt and yespower have been designed by Solar Designer building upon scrypt.

The following other people and projects have also indirectly helped make yespower what it is:

- Bill Cox
- Rich Felker
- Anthony Ferrara
- Christian Forler
- Taylor Hornby
- Dmitry Khovratovich
- Samuel Neves
- Marcos Simplicio
- Ken T Takusagawa

 - Jakob Wenzel
 - Christian Winnerlein

 - DARPA Cyber Fast Track
 - Password Hashing Competition

Contact info.

First, please check the yespower homepage for new versions, etc.:

    http://www.openwall.com/yespower/

If you have anything valuable to add or a non-trivial question to ask,
you may contact the maintainer of yespower at:

    Solar Designer <solar at openwall.com>