---

title: Dagger-Hashimoto description: A detailed look at the Dagger-Hashimoto algorithm. lang: en

---

Dagger-Hashimoto was the original research implementation and specification for Ethereum's mining algorithm. Dagger-Hashimoto was superseded by Ethash. Mining was switched off completely atThe Merge on 15th September 2022. Since then, Ethereum has been secured using a proof-of-stake mechanism instead. This page is for historical interest - the information here is no longer relevant for post-Merge Ethereum.

Prerequisites {#prerequisites}

To better understand this page, we recommend you first read up onproof-of-work consensus, mining, and mining algorithms.

Dagger-Hashimoto {#dagger-hashimoto}

Dagger-Hashimoto aims to satisfy two goals:

1. **ASIC-resistance**: the benefit from creating specialized hardware for the algorithm should be as small as possible
2. **Light client verifiability**: a block should be efficiently verifiable by a light client.

With an additional modification, we also specify how to fulfill a third goal if desired, but at the cost of additional complexity:

Full chain storage: mining should require storage of the complete blockchain state (due to the irregular structure of the Ethereum state trie, we anticipate that some pruning will be possible, particularly of some often-used contracts, but we want to minimize this).

DAG Generation {#dag-generation}

The code for the algorithm will be defined in Python below. First, we give`encode_int` for marshaling unsigned ints of specified precision to strings. Its inverse is also given:

```python NUM_BITS = 512

def encode_int(x): "Encode an integer x as a string of 64 characters using a big-endian scheme" o = '' for _ in range(NUM_BITS / 8): o = chr(x % 256) + o x //= 256 return o

def decode_int(s): "Unencode an integer x from a string using a big-endian scheme" x = 0 for c in s: x *= 256 x += ord(c) return x ```

We next assume that `sha3` is a function that takes an integer and outputs an integer, and`dbl_sha3` is a double-sha3 function; if converting this reference code into an implementation use:

```python from pyethereum import utils def sha3(x): if isinstance(x, (int, long)): x = encode_int(x) return decode_int(utils.sha3(x))

def dbl_sha3(x): if isinstance(x, (int, long)): x = encode_int(x) return decode_int(utils.sha3(utils.sha3(x))) ```

### Parameters {#parameters}

The parameters used for the algorithm are:

```python SAFE_PRIME_512 = 2**512 - 38117 # Largest Safe Prime less than 2**512

params = { "n": 4000055296 * 8 // NUM_BITS, # Size of the dataset (4 Gigabytes); MUST BE MULTIPLE OF 65536 "n_inc": 65536, # Increment in value of n per period; MUST BE MULTIPLE OF 65536 # with epochtime=20000 gives 882 MB growth per year "cache_size": 2500, # Size of the light client's cache (can be chosen by light # client; not part of the algo spec) "diff": 2**14, # Difficulty (adjusted during block evaluation) "epochtime": 100000, # Length of an epoch in blocks (how often the dataset is updated) "k": 1, # Number of parents of a node "w": w, # Used for modular exponentiation hashing "accesses": 200, # Number of dataset accesses during hashimoto "P": SAFE_PRIME_512 # Safe Prime for hashing and random number
```

generation } ```

`P` in this case is a prime chosen such that $\log_2(P)$ is just slightly less than 512, which corresponds to the 512 bits we have been using to represent our numbers. Note that only the latter half of the DAG actually needs to be stored, so the de-facto RAM requirement starts at 1 GB and grows by 441 MB per year.

Dagger graph building {#dagger-graph-building}

The dagger graph building primitive is defined as follows:

```
python def produce_dag(params, seed, length): P = params["P"] picker = init = pow(sha3(seed), params["w"], P) o
= [init] for i in range(1, length): x = picker = (picker * init) % P for _ in range(params["k"]): x ^= o[x % i]
o.append(pow(x, params["w"], P)) return o
```

Essentially, it starts off a graph as a single node, `sha3(seed)`, and from there starts sequentially adding on other nodes based on random previous nodes. When a new node is created, a modular power of the seed is computed to randomly select some indices less than `i` (using `x % i` above), and the values of the nodes at those indices are used in a calculation to generate a new a value for `x`, which is then fed into a small proof of work function (based on XOR) to ultimately generate the value of the graph at index `i`. The rationale behind this particular design is to force sequential access of the DAG; the next value of the DAG that will be accessed cannot be determined until the current value is known. Finally, modular exponentiation hashes the result further.

This algorithm relies on several results from number theory. See the appendix below for a discussion.

Light client evaluation {#light-client-evaluation}

The above graph construction intends to allow each node in the graph to be reconstructed by computing a subtree of only a small number of nodes and requiring only a small amount of auxiliary memory. Note that with k=1, the subtree is only a chain of values going up to the first element in the DAG.

The light client computing function for the DAG works as follows:

```python def quick_calc(params, seed, p): w, P = params["w"], params["P"] cache = {}

```
def quick_calc_cached(p):
    if p in cache:
        pass
    elif p == 0:
        cache[p] = pow(sha3(seed), w, P)
    else:
        x = pow(sha3(seed), (p + 1) * w, P)
        for _ in range(params["k"]):
            x ^= quick_calc_cached(x % p)
        cache[p] = pow(x, w, P)
    return cache[p]

return quick_calc_cached(p)
```

```

Essentially, it is simply a rewrite of the above algorithm that removes the loop of computing the values for the entire DAG and replaces the earlier node lookup with a recursive call or a cache lookup. Note that for `k=1` the cache is unnecessary, although a further optimization actually precomputes the first few thousand values of the DAG and keeps that as a static cache for computations; see the appendix for a code implementation of this.

Double buffer of DAGs {#double-buffer}

In a full client, a *double buffer* of 2 DAGs produced by the above formula is used. The idea is that DAGs are produced every `epochtime` number of blocks according to the params above. Instead of the client using the latest DAG produced, it uses the one previous. The benefit of this is that it allows the DAGs to be replaced over time without needing to incorporate a step where miners must suddenly recompute all of the data. Otherwise, there is the potential for an abrupt temporary slowdown in chain processing at regular intervals and dramatically increasing centralization. Thus 51% attack risks within those few minutes before all data are recomputed.

The algorithm used to generate the set of DAGs used to compute the work for a block is as follows:

~~~
```python def get_prevhash(n): from pyethereum.blocks import GENESIS_PREVHASH from pyethereum import
chain_manager if num <= 0: return hash_to_int(GENESIS_PREVHASH) else: prevhash =
chain_manager.index.get_block_by_number(n - 1) return decode_int(prevhash)

def get_seedset(params, block): seedset = {} seedset["back_number"] = block.number - (block.number %
params["epochtime"]) seedset["back_hash"] = get_prevhash(seedset["back_number"]) seedset["front_number"] =
max(seedset["back_number"] - params["epochtime"], 0) seedset["front_hash"] = get_prevhash(seedset["front_number"])
return seedset

def get_dagsize(params, block): return params["n"] + (block.number // params["epochtime"]) * params["n_inc"]

def get_daggerset(params, block): dagsz = get_dagsize(params, block) seedset = get_seedset(params, block) if
seedset["front_hash"] <= 0: # No back buffer is possible, just make front buffer return {"front": {"dag": produce_dag(params,
seedset["front_hash"], dagsz), "block_number": 0}} else: return {"front": {"dag": produce_dag(params, seedset["front_hash"],
dagsz), "block_number": seedset["front_number"]}, "back": {"dag": produce_dag(params, seedset["back_hash"], dagsz),
"block_number": seedset["back_number"]}} ```
~~~

Hashimoto {#hashimoto}

The idea behind the original Hashimoto is to use the blockchain as a dataset, performing a computation that selects N indices from the blockchain, gathers the transactions at those indices, performs an XOR of this data, and returns the hash of the result. Thaddeus Dryja's original algorithm, translated to Python for consistency, is as follows:

```
python def orig_hashimoto(prev_hash, merkle_root, list_of_transactions, nonce): hash_output_A = sha256(prev_hash
+ merkle_root + nonce) txid_mix = 0 for i in range(64): shifted_A = hash_output_A >> i transaction = shifted_A %
len(list_of_transactions) txid_mix ^= list_of_transactions[transaction] << i return txid_max ^ (nonce << 192)
```

Unfortunately, while Hashimoto is considered RAM hard, it relies on 256-bit arithmetic, which has considerable computational overhead. However, Dagger-Hashimoto only uses the least significant 64 bits when indexing its dataset to address this issue.

```
python def hashimoto(dag, dagsize, params, header, nonce): m = dagsize / 2 mix = sha3(encode_int(nonce) +
header) for _ in range(params["accesses"]): mix ^= dag[m + (mix % 2**64) % m] return dbl_sha3(mix)
```

The use of double SHA3 allows for a form of zero-data, near-instant pre-verification, verifying only that a correct intermediate value was provided. This outer layer of proof-of-work is highly ASIC-friendly and fairly weak, but exists to make DDoS even more difficult since that small amount of work must be done in order to produce a block that will not be rejected immediately. Here is the light-client version:

```
python def quick_hashimoto(seed, dagsize, params, header, nonce): m = dagsize // 2 mix = sha3(nonce + header)
for _ in range(params["accesses"]): mix ^= quick_calc(params, seed, m + (mix % 2**64) % m) return dbl_sha3(mix)
```

Mining and verifying {#mining-and-verifying}

Now, let us put it all together into the mining algorithm:

```
python def mine(daggerset, params, block): from random import randint nonce = randint(0, 2**64) while 1: result
= hashimoto(daggerset, get_dagsize(params, block), params, decode_int(block.prevhash), nonce) if result *
params["diff"] < 2**256: break nonce += 1 if nonce >= 2**64: nonce = 0 return nonce
```

Here is the verification algorithm:

```
python def verify(daggerset, params, block, nonce): result = hashimoto(daggerset, get_dagsize(params, block),
params, decode_int(block.prevhash), nonce) return result * params["diff"] < 2**256
```

Light-client friendly verification:

```
python def light_verify(params, header, nonce): seedset = get_seedset(params, block) result =
quick_hashimoto(seedset["front_hash"], get_dagsize(params, block), params, decode_int(block.prevhash), nonce)
return result * params["diff"] < 2**256
```

Also, note that Dagger-Hashimoto imposes additional requirements on the block header:

- For two-layer verification to work, a block header must have both the nonce and the middle value pre-sha3
- Somewhere, a block header must store the sha3 of the current seedset

Further reading {#further-reading}

Know of a community resource that helped you? Edit this page and add it!

Appendix {#appendix}

As noted above, the RNG used for DAG generation relies on some results from number theory. First, we provide assurance that the Lehmer RNG that is the basis for the `picker` variable has a wide period. Second, we show that `pow(x,3,P)` will not map `x` to `1` or `P-1` provided `x ∈ [2,P-2]` to start. Finally, we show that `pow(x,3,P)` has a low collision rate when treated as a hashing function.

Lehmer random number generator {#lehmer-random-number}

While the `produce_dag` function does not need to produce unbiased random numbers, a potential threat is that `seed**i % P` only takes on a handful of values. This could provide an advantage to miners recognizing the pattern over those that do not.

To avoid this, a result from number theory is appealed to. A *Safe Prime* is defined to be a prime `P` such that `(P-1)/2` is also prime. The *order* of a member `x` of the multiplicative group `ℤ/nℤ` is defined to be the minimal `m` such that

$x^m \bmod P \equiv 1$

Given these definitions, we have:

> Observation 1. Let `x` be a member of the multiplicative group `ℤ/Pℤ` for a safe prime `P`. If `x mod P ≠ 1 mod P` and `x mod P ≠ P-1 mod P`, then the order of `x` is either `P-1` or `(P-1)/2`.

Proof. Since `P` is a safe prime, then by [Lagrange's Theorem][lagrange] we have that the order of `x` is either `1`, `2`, `(P-1)/2`, or `P-1`.

The order of `x` cannot be `1`, since by Fermat's Little Theorem we have:

$x^{P-1} \bmod P \equiv 1$

Hence `x` must be a multiplicative identity of `ℤ/nℤ`, which is unique. Since we assumed that `x ≠ 1` by assumption, this is not possible.

The order of `x` cannot be `2` unless `x = P-1`, since this would violate that `P` is prime.

From the above proposition, we can recognize that iterating `(picker * init) % P` will have a cycle length of at least `(P-1)/2`. This is because we selected `P` to be a safe prime approximately equal to be a higher power of two, and `init` is in the interval `[2,2**256+1]`. Given the magnitude fo `P`, we should never expect a cycle from modular exponentiation.

When we are assigning the first cell in the DAG (the variable labeled `init`), we compute `pow(sha3(seed) + 2, 3, P)`. At first glance, this does not guarantee that the result is neither `1` nor `P-1`. However, since `P-1` is a safe prime, we have the following additional assurance, which is a corollary of Observation 1:

> Observation 2. Let `x` be a member of the multiplicative group `ℤ/Pℤ` for a safe prime `P`, and let `w` be a natural number. If `x mod P ≠ 1 mod P` and `x mod P ≠ P-1 mod P`, as well as `w mod P ≠ P-1 mod P` and `w mod P ≠ 0 mod P`, then x^w `mod P ≠ 1 mod P` and x^w `mod P ≠ P-1 mod P`

Modular exponentiation as a hash function {#modular-exponentiation}

For certain values of `P` and `w`, the function `pow(x, w, P)` may have many collisions. For instance, `pow(x,9,19)` only takes on values `{1,18}`.

Given that `P` is prime, then an appropriate `w` for a modular exponentiation hashing function can be chosen using the following result:

Observation 3. Let P be a prime; w and P-1 are relatively prime if and only if for all a and b in ℤ/Pℤ:

$$a^w \bmod P \equiv b^w \bmod P \text{ if and only if } a \bmod P \equiv b \bmod P$$

Thus, given that P is prime and w is relatively prime to P-1, we have that |{pow(x, w, P) : x ∈ ℤ}| = P , implying that the hashing function has the minimal collision rate possible.

In the special case that P is a safe prime as we have selected, then P-1 only has factors 1, 2, (P-1)/2 and P-1. Since P > 7, we know that 3 is relatively prime to P-1, hence w=3 satisfies the above proposition.

More efficient cache-based evaluation algorithm {#cache-based-evaluation}

```python def quick_calc(params, seed, p): cache = produce_dag(params, seed, params["cache_size"]) return quick_calc_cached(cache, params, p)

def quick_calc_cached(cache, params, p): P = params["P"] if p < len(cache): return cache[p] else: x = pow(cache[0], p + 1, P) for _ in range(params["k"]): x ^= quick_calc_cached(cache, params, x % p) return pow(x, params["w"], P)

def quick_hashimoto(seed, dagsize, params, header, nonce): cache = produce_dag(params, seed, params["cache_size"]) return quick_hashimoto_cached(cache, dagsize, params, header, nonce)

def quick_hashimoto_cached(cache, dagsize, params, header, nonce): m = dagsize // 2 mask = 2**64 - 1 mix = sha3(encode_int(nonce) + header) for _ in range(params["accesses"]): mix ^= quick_calc_cached(cache, params, m + (mix & mask) % m) return dbl_sha3(mix) ```
```