A Formalization of Signum's Consensus

anonymous author

anonymous institute
anonymous email

Abstract. Bitcoin's proof of work consensus consumes energy and requires dedicated, expensive hardware. Therefore, alternatives have been proposed, including proof of stake and proof of space. The latter mines with disk space instead of CPU power. Signum is the only implemented proof of space blockchain with smart contracts, and runs since ten years. But its relatively simple consensus algorithm lacks any formalization. This paper formalizes Signum's consensus and uses that formalization to show that Signum is free from block grinding attacks and is largely protected from challenge grinding attacks. Moreover, this paper proposes a new protection for Signum against newborn attacks.

1 Introduction

A blockchain is a list of *blocks*, each reporting the hash of a previous block, satisfying some consistency or *consensus* rules. Blocks hold *transactions*, whose exact nature is not relevant here: they are requests to update the state of a global abstract machine (a ledger of payments as in Bitcoin [15,2]) or a sort of global memory where data structures are allocated and modified (as in Ethereum [3]). By using hashes as machine-independent pointers, blockchains can be distributed in a network of peers. This is desirable since data gets safely duplicated and no special peer determines the history by itself. However, peers expand the blockchain, independently from other peers, hence the blockchain becomes a tree rather than a list. A notion of chain quality incentivizes peers to append blocks to the highest-quality chain (the *best* chain). Therefore, a peer could replace its current best chain with another, even better chain, a so called *history change*.

As presented above, peers are free to generate new blocks at maximal speed, flood the network with new blocks and make the emergence of a best chain difficult. This is an efficiency and security issue: frequent history changes allow *double spending* and network forks. The actual genious of Nakamoto [15] was to (largely) solve this issue with a consensus rule requiring blocks to answer a *challenge* contained in their previous block. Namely, the hash of each block must be smaller than a *difficulty* value computed from the previous blocks, directly bound to the quality of the chain. Therefore, who creates (*mines*) a new block runs a *proof of work* algorithm that rotates (*grinds*) many alternative values for a block field (*nonce*), until the hash of the block is smaller than the difficulty. This hardens the creation of new blocks, makes it impossible to create blocks at arbitrary speed and introduces an incentive to expanding the best chain only,

rather than creating alternative histories by mining on multiple chains. The difficulty changes overtime, to account for change in the total hashing power of the network. As shown in [9], this stabilizes the block creation rate and supports network consistency (all honest peers converge to the same chain, eventually).

Proof of work is a brute-force algorithm, because of the non-correlation property of hash functions. Therefore, it consumes energy (as much as a medium-sized country, for Bitcoin); moreover, it is not egalitarian, being worthwhile only in countries where electricity is cheap; furthermore, it is more efficient over dedicated, expensive hardware (such as ASICs), against the promise of a democratic and open network. Therefore, the current trend is towards *proof of stake*. Its different flavors share the common idea that mining is limited to a (static or dynamic, exclusive or delegatable) set of peers (*validators*), that *stake* a collateral in exchange of mining rights. Many criticize proof of stake for being centralized and undemocratic (*rich becomes richer*). Moreover, it suffers from what we call a *start-up issue*: as long as the cryptocurrency of a newborn blockchain has still no value, validators have no incentive to work and be updated. Moreover, validators get punished (*slashed*) if they misbehave or are offline, which might be perceived as unfair if that happens because of a connectivity issue or black-out.

A further alternative is *proof of space* [4,8], where miners must dedicate a large chunk of disk memory for answering challenges. Its energy consumption is negligeable and no special hardware helps for mining, currently: the technology is both cheap and democratic. Moreover, proof of space allows one to capitalize on unused memory, for free, while proof of work has always an inherent electricity cost. For fairness, proof of space protocols should only allow to generate answers of quality directly proportional to the allocated space, or otherwise they are said to suffer from a time/memory tradeoff. As a drawback, cheap answers introduce new *nothing-at-stake* security attacks [16], that are instead anti-economic with proof of work, since computing power can only be dedicated to one mining task:

Block grinding: Miners might find it profitable (*rational*) to mine many alternative new blocks, each holding different transactions, finally selecting the block that leads to better answers to subsequent challenges.

Challenge grinding: Miners might find it profitable to provide suboptimal answers to the current challenge if this leads to subsequent challenges for which they have much better answers.

Mining on multiple chains: Miners might find it profitable to mine multiple chains simultaneously (not only the best one).

These attacks increase the risk of double spending and make it convenient to mine through space *and* work, thus neutralizing the benefits of proof of space.

Another nothing-at-stake problem, that has not received great attention up to now, is the *newborn attack* [24]: a miner that has allocated a large space for mining for a blockchain network N could use the same space, unchanged, for mining for a newborn, small network N'. If the total space used by the peers of N' is initially relatively small, it could be possible for the miner to hijack the history of N', effectively taking its control.

Most formalizations of proof of space [4,8,18] are based on challenges against graphs of high pebbling complexity, but no actual blockchain has ever been built that way: only SpaceMint [16] exists which is not a real blockchain but a minimal non-maintained prototype of the theoretical consensus protocol only. Namely, a real graph pebbling blockchain has never been implemented because: (1) the protocol includes an *initialization phase*, run for each new prover (miner) that joins the blockchain, that complicates the protocol itself and requires to spend cryptocurrency *before* starting mining; (2) answers to challenges (*proofs*), included in blockchain, are relatively large [1]: kilobytes or even megabytes for proofs created in the initialization phase.

An alternative is Permacoin [13], based on proof of retrievability rather than graph pebbling, still in the family of the proof of space algorithms [18]. But neither Permacoin has been implemented: only a prototypical and minimal implementation of only its consensus algorithm exists, discontinued in 2014. Neither SpaceMint nor Permacoin have been shown to support smart contracts.

Our target of analysis has been Signum [20], instead, since we wanted a fully-fledged implementation, which gives practical relevance to our work. Signum (previously Burstcoin) is actually deployed and runs continously since 2014. It provides smart contracts on top of its consensus algorithms. Signum is not based on graph pebbling: instead, each miner precomputes a large *plot file* of hashes, that is not shared nor stored in blockchain. A peer that wants to mine the next block derives a challenge from the current blockchain head and challenges a miner for an answer, called *deadline*, *ie.*, a small (less than 200 bytes) data structure, that can be very quickly derived from the plot, with a quality measure (its *waiting time*) proportional, on average, to the size of the plot. Signum's protocol is attractive since it has no initialization phase (each miner creates its plot file independently and off-line) and its answers are very small. In principle, Signum can be mined as described above (proof of space) but also by recomputing the plot file on-the-fly, at each challenge, without any disk allocation (proof of work). Because of that, Signum's mining is sometimes called *proof of capacity*. However, the proof of work version of Signum remains theoretical and there is no evidence that Signum has ever been mined like that. This is because plots use an expensive hashing algorithm (shabal256) and are very big, so that their recomputation takes longer than the block creation rate. Specialized hardware might change the situation in the future but, from its inception in 2014 to the present day, Signum seems to have been only mined with proof of space.

The actual drawback of Signum is that the underlying theory has never been formalized nor defined up to now. Moroever, [16] warned about a potential block grinding attack. Without a formalization, it is impossible to judge if it is real.

Therefore, this paper provides the following contributions about Signum:

- a formalization of its algorithm, recostructed and interpolated from a very informal and partial description [22] and its poorly commented code [21];
- a proof that block grinding attacks are impossible, against a previous hint [16];
- a proof that challenge grinding attacks are limited;
- a new protection against newborn attacks.

notation	meaning	in [22]
$\bowtie$	concatenation of sequences	
$\#scoops$	number of scoops contained in a nonce	4096
$h_{deadline}$	hashing function for computing nonces, plots and deadlines	shabal256
$h_{generation}$	hashing function for computing the generations of challenges	shabal256
h_{block}	hashing function for computing the hash of the blocks	sha256
κ	threshold to the number of bytes fed to $h_{deadline}$ in Alg. 1	4096
$beat$	target block creation time interval (ms)	240000
$\sigma_{genesis}$	generation signature for the genesis block	
τ_{now}	current time (ms from Unix epoch)	
$oblivion$	acceleration reaction to changes of mining power (0 to 1)	

Table 1. Notations and contextual information used in our formalization and their specific instantiations used in [22], when available.

These results are relevant since they show that Signum's consensus is actually supported by a formal theory and protected from a large class of attacks.

The rest of this paper is organized as follows. Sec. 2 formalizes the structure of the plot files. Sec. 3 defines the challenges that the consensus algorithm must solve, and their answers (deadlines). Sec. 4 presents Signum's mining algorithm. Sec. 5 studies grinding attacks in Signum and proposes a new solution against newborn attacks. Sec. 6 presents related work. Sec. 7 concludes. Proofs are reported in appendix. Tab. 1 collects notations used throughout the paper. It also reports specific choices made in [22] (rightmost column), but this paper remains parametric *wrt.* them. For instance, for genericity, our formalization uses three hashing functions, that might actually coincide.

Acknowledgments: Removed for anonymization.

2 Nonces and Plots

This section formalizes the notions of nonce and plot and their algorithmic construction. Namely, Signum requires miners to hold one or more plots (sets of nonces) on disk. Their initialization is performed only once and offline, hence it is not part of the mining protocol.

The following definitions are used to deal with bytes and hashing.

Definition 1 (Concatenation operator $\bowtie$). *Sequences (for instance, of bytes) are concatenated by $\bowtie$. The same $\bowtie$ is used to concatenate a sequence to an element or an element to a sequence.*

The following byte representation of natural numbers is used in [22]. It is the standard representation in most computers nowadays.

Definition 2 (*nat2be* and *be2nat*). *The operators nat2be and be2nat transform natural numbers into their big-endian byte representation, and vice versa.*

We recall that the big-endian representation of a natural number is its binary representation, split in bytes, with the most significant byte placed first.

Definition 3 (Hashing function). *A* hashing function h *of size* > 0 *is a total map* $h : byte^* \to byte^{size}$*, where* $byte^*$ *is a sequence of bytes, of arbitrary length, and* $byte^{size}$ *is a sequence of size bytes, called a* hash *for* h*. If* h *is a hashing function, then* $size(h)$ *is its size.*

A *scoop* is a pair of hashes. A *nonce* is a natural *progressive number* p, and a list of $\#scoops > 0$ scoops. Their definitions are parametric *wrt.* a hashing function $h_{deadline}$ used for their creation.

Definition 4 (Scoop, Nonce). *The sets of* scoops *and* nonces *are*

$$\mathsf{Scoops} = \left\{ \langle h_1, h_2 \rangle \,\middle|\, h_1, h_2 \text{ are hashes for } h_{deadline} \right\},$$
$$\mathsf{Nonces} = \left\{ \langle p, scoops \rangle \,\middle|\, p \in \mathbb{N} \text{ and } scoops \in \mathsf{Scoops}^{\#scoops} \right\}.$$

In the above definition, angular brackets stand for tuples (in this specific case, they stand for pairs). When definitions are given in terms of tuples, they silently introduce selection functions for the tuple elements. For instance, if $nonce \in \mathsf{Nonces}$, then $nonce.p$ and $nonce.scoops$ are its elements.

A *prolog* is the identifier of the creator of nonces and plots (for instance, its public key). For now, it is just a sequence of bytes. Sec. 5 will give structure to prologs and see how they can be useful.

Definition 5 (Prolog). *The set of* prologs *is* $\mathsf{Prologs} = \{\pi \mid \pi \in bytes^*\}$.

Algorithm 1 constructs a nonce, given its progressive number and a prolog. It uses a constant $\kappa > 0$ (Tab. 1) to limit its computational cost, to avoid hashing very large chunks of data. This algorithm derives a sequence of hashes (steps 1 and 2) and constructs a *final* hash from all of them (step 3) that uses to modify the original sequence of hashes (step 4). Then it shuffles the sequence (step 5), which intuitively guarantees that, in order to compute a given scoop of the nonce (*ie.* a pair of consecutive hashes), the algorithm must be thoroughly executed.

Algorithm 1 (*nonce*(p, π)) *Given* $p \in \mathbb{N}$ *and* $\pi \in \mathsf{Prologs}$*, we define*

$$nonce(p, \pi) = \langle p, \langle h_0, h_1 \rangle \bowtie \cdots \bowtie \langle h_{2 \cdot \#scoops - 2}, h_{2 \cdot \#scoops - 1} \rangle \rangle \in \mathsf{Nonces},$$

where the hashes $h_0, \ldots, h_{2 \cdot \#scoops - 1}$ *are constructed as follows*[1]*.*

1. *Let* $seed = \pi \bowtie nat2be(p)$.
2. *For each* i *from* $2 \cdot \#scoops - 1$ *to* 0*, let*[2]

$$h_i = h_{deadline}\left(\textit{first } \kappa \textit{ bytes of} \left(\left(\bowtie_{i<j<2 \cdot \#scoops} h_j \right) \bowtie seed \right) \right).$$

[1] Step. 5 and the threshold κ have been added after the publication of [16], in response to some of their criticisms. See Sec. 5.

[2] In [22], it is said to take the *last* κ bytes, that would be meaningless since then the lowest h_i's would coincide. An inspection of their code shows that they actually take the *first* κ bytes. They probably use *last* here in the sense of *more recently computed*.

3. *Let* $h_{final} = h_{deadline}\left((\bowtie_{0 \le j < 2 \cdot \#scoops} h_j) \bowtie seed\right)$.
4. *For each* i *from* 0 *to* $2 \cdot \#scoops - 1$, *reassign* h_i *to* $h_i \oplus h_{final}$.
5. *For each odd* i *from* 1 *to* $2 \cdot \#scoops - 1$, *swap* h_i *with* $h_{2 \cdot \#scoops - i}$.

A *plot* is a set of nonces constructed with Alg. 1, for a finite non-empty set of progressive numbers P and for a given prolog π, recorded in the plot.

Definition 6 (Plot). *The set of* plots *is defined as*

$$\mathsf{Plots} = \left\{ \langle \pi, nonces \rangle \;\middle|\; \begin{array}{l} \pi \in \mathsf{Prologs},\ \varnothing \neq P \subset \mathbb{N} \text{ is finite} \\ \text{and } nonces = \{ nonce(p, \pi) \mid p \in P \} \end{array} \right\}.$$

The computations of $nonce(p, \pi)$ and $nonce(p', \pi)$, for $p \neq p'$, are completely independent. Therefore, Def. 6 implies that the construction of a plot can be optimized on multicore hardware.

3 Challenges and Deadlines

A *challenge* specifies a puzzle that must be solved in order to mine a new block. In Signum, challenges become a query that can be asked to each nonce of a plot, resulting in an answer called *deadline*.

Definition 7 (Challenge). *The set of* challenges *is*

$$\mathsf{Challenges} = \left\{ \langle scoopNumber, \sigma \rangle \;\middle|\; \begin{array}{l} 0 \le scoopNumber < \#scoops \\ \text{and } \sigma \text{ is a hash for } h_{generation} \end{array} \right\}.$$

The σ *component of a challenge is said to be its* generation signature. *In the following, generation signature will be used as a synonym of hash for* $h_{generation}$.

Given a challenge and a nonce, the latter has a value that specifies how well the nonce answers the challenge.

Definition 8 ($value(nonce, challenge)$**).** *Let* $nonce \in \mathsf{Nonces}$ *and* $challenge \in \mathsf{Challenges}$. *The* value *of nonce wrt. challenge, ie.,* $value(nonce, challenge)$, *is* $h_{deadline}(nonce.scoops[challenge.scoopNumber] \bowtie challenge.\sigma)$.

The answer to a challenge could actually be a nonce n, whose quality is its value. But nonces are quite big (around 262 kbytes under the assumptions in the rightmost column of Tab. 1). Since answers are stored in blockchain, [22] introduces *deadlines*, a much smaller representation of the value of n, carrying the information needed to reconstruct n and verify that it actually answers the challenge.

Definition 9 (Deadline). *The set of* deadlines *is*

$$\mathsf{Deadlines} = \left\{ \langle p, \pi, value, challenge \rangle \;\middle|\; \begin{array}{l} \pi \in \mathsf{Prologs},\ p \in \mathbb{N}, \\ value \text{ is a hash for } h_{deadline} \\ \text{and } challenge \in \mathsf{Challenges} \end{array} \right\}.$$

Deadlines are totally ordered by increasing value.

Intuitively, the value of a deadline expresses how many milliseconds must be waited until the deadline expires and a new block can be mined. However, if the mining power of the network increases, the minimal value of the deadlines generated by the network tends to decrease and the block creation rate would not be fixed to *beat* (Tab. 1), on average. This explains why the deadlines' value is modulated *wrt.* an *acceleration*[3], which is the inverse of Bitcoin's difficulty.

Definition 10 (Deadline's waiting time). *Given* $\delta \in$ Deadlines *and an* acceleration $\alpha \in \mathbb{N}$ *such that* $\alpha > 0$*, the* waiting time *for* δ *wrt.* α *is*[4]

$$waitingTime(\delta, \alpha) = \frac{be2nat(\delta.value)}{\alpha}.$$

Def. 11 finally shows how a nonce answers a challenge with a deadline.

Definition 11 ($\delta(nonce, \pi, challenge)$). *Given* $nonce \in$ Nonces*,* $\pi \in$ Prologs *and* $challenge \in$ Challenges*, the* deadline computed from *nonce* for π and *challenge is* $\delta(nonce, \pi, challenge) = \langle nonce.p, \pi, value(nonce, challenge), challenge \rangle$.

Def. 11 extends to plots. Remember that plots are non-empty (Def. 6) and embed the identifier of their creator π; and that deadlines are ordered by their value.

Definition 12 ($deadline(plot, challenge)$). *Given* $plot \in$ Plots *and* $challenge \in$ Challenges*, the* deadline computed from *plot* for *challenge is*[5]

$$\delta(plot, challenge) = \min_{nonce \in plot.nonces} \delta(nonce, plot.\pi, challenge).$$

A deadline is valid when the nonce built from its progressive and prolog has the same value as the deadline *wrt.* its challenge.

Definition 13 (Deadline's validity). *Given* $\delta \in$ Deadlines*, it is* valid *if and only if* $\delta.value = value(nonce(\delta.p, \delta.\pi), \delta.challenge)$.

4 Blockchain Construction

The blocks of the blockchain contain information used for consensus, called *trunk* by borrowing this terminology from [6]; other information such as the previous block hash; and extra information that is irrelevant here, such as a list of transactions, that are not formalized below since they are not used by Signum's consensus. Blocks can be genesis and non-genesis. Both contain their time of creation and their acceleration. Genesis blocks have no trunk nor parent; their height is implicitly 0. Challenges c are generated in sequence: there is an initial constant challenge for the genesis block, while the challenge of non-genesis

[3] In [22] the term *base target* is used for it, but we think that *acceleration* is clearer.

[4] In [22], the divisor is actually $2^{size(h_{deadline})-8} \cdot \alpha$, to avoid using very large values for α. This is theoretically irrelevant and we prefer our simpler presentation.

[5] If more nonces of the plot lead to deadlines with the same minimal value, we assume that Def. 12 chooses one, according to some policy that is irrelevant here.

blocks b is generated from their trunk. In particular, c is *not* computed from the transactions in b, in order to avoid block-grinding attacks (Sec. 5). A deadline that answers c is recorded in the trunk of the sons of b.

Definition 14 (Trunk, Block). *The sets of* trunks *and* blocks *are*

$$\begin{aligned}
\mathsf{Trunks} &= \{\langle \mathit{height}, \delta\rangle \mid \mathit{height} \in \mathbb{N} \text{ and } \delta \in \mathsf{Deadlines}\},\\
\mathsf{GenesisBlocks} &= \{\langle \tau, \alpha\rangle \mid \tau \in \mathbb{N},\ \alpha \in \mathbb{N} \text{ and } \alpha > 0\},\\
\mathsf{NonGenesisBlocks} &= \left\{ \left\langle \begin{array}{c} \tau, \alpha, \mathit{power},\\ \mathit{weightedBeat}, \mathit{trunk},\\ \mathit{previousBlockHash} \end{array} \right\rangle \middle| \begin{array}{l} \tau, \alpha \in \mathbb{N},\ \alpha > 0,\\ \mathit{power}, \mathit{weightedBeat} \in \mathbb{N},\\ \mathit{trunk} \in \mathsf{Trunks},\\ \mathit{previousBlockHash} \text{ is a}\\ \quad \mathit{hash for } h_{\mathit{block}} \end{array} \right\},\\
\mathsf{Blocks} &= \mathsf{GenesisBlocks} \cup \mathsf{NonGenesisBlocks}.
\end{aligned}$$

If b is a block, then $b.\tau$ is its creation time (milliseconds from the Unix epoch) and $b.\alpha$ is its acceleration. If b is a non-genesis block, then $b.\mathit{power}$ expresses how much space has been used to build the path to b, starting from the genesis block; it will be used to select the *best chain* for mining b's sons. The value of $b.\mathit{weightedBeat}$ is the average block creation rate in the path to b; it weighs the last blocks more. It will be compared to *beat* (Tab. 1) to understand if the acceleration must be increased or decreased in b's sons. The value of $b.\mathit{previousBlockHash}$ is the hash of the previous block in the path to b. If b is a genesis block, we abuse notation and assume that $b.\mathit{power} = b.\mathit{weightedBeat} = 0$.

Definition 15 (Block's height). *Let* $b \in \mathsf{Blocks}$. *The* height *of* b*, written* $\mathit{height}(b)$*, is* 0 *if* $b \in \mathsf{GenesisBlocks}$*, and* $b.\mathit{trunk}.\mathit{height}$ *if* $b \in \mathsf{NonGenesisBlocks}$.

Def. 16 shows how the first challenge is defined, for genesis blocks. It is a constant that only depends on contextual values (Table 1).

Definition 16 (*initialChallenge*). *The* initial challenge *is*[6] $\mathit{initialChallenge} = \langle 0, \sigma_{\mathit{genesis}}\rangle$*, where* $\sigma_{\mathit{genesis}}$ *is a constant generation signature used for the genesis of the blockchain (see Table 1).*

Def. 17 shows how a challenge is derived from the trunk of a non-genesis block.

Definition 17 (*challenge*$_{\mathit{next}}$(*trunk*)). *Let* $\mathit{trunk} \in \mathsf{Trunks}$. *The* next challenge for *trunk is* $\mathit{challenge}_{\mathit{next}}(\mathit{trunk}) = \langle \mathit{be2nat}(h_{\mathit{generation}}(\sigma \bowtie \mathit{nat2be}(\mathit{trunk}.\mathit{height}+1))) \bmod \#\mathit{scoops}, \sigma\rangle$*, where* $\sigma = h_{\mathit{generation}}(\mathit{trunk}.\delta.\mathit{challenge}.\sigma \bowtie \mathit{trunk}.\delta.\pi)$.

The construction of the generation signature σ for the next challenge, in Def. 17, has puzzled us for some time, since [22] appends a *previous block generator* to the previous block's generation signature $\mathit{trunk}.\delta.\mathit{challenge}.\sigma$. That concept, however, is defined nowhere. We had to dive in the source code of Signum to

[6] In [22] the scoop number is derived from $\sigma_{\mathit{genesis}}$; we simplify it to 0, without loss of generality.

understand that it is actually an identifier (more concretely, the public key) of the creator of the deadline for the previous block (see `https://github.com/signum-network/signum-node/blob/main/src/brs/GeneratorImpl.java`, in the constructor of `GeneratorStateImpl`). Our prolog of the previous deadline generalizes that information, hence Def. 17 appends $trunk.\delta.\pi$ to define σ.

Later, it will be handy to determine the next challenge for a block. Note that it only uses the trunk inside the block.

Definition 18. *Let* $b \in \mathsf{Blocks}$*. Its* next challenge *is*

$$challenge_{next}(b) = \begin{cases} initialChallenge & \text{if } b \in \mathsf{GenesisBlocks} \\ challenge_{next}(b.trunk) & \text{if } b \in \mathsf{NonGenesisBlocks}. \end{cases}$$

Def. 19 shows how the information inside a block is used to construct that inside its sons. It is actually our proposal, since there is no information in [22] about this. The computation of the next weighted beat gives more or less weight to the previous weighted beat, depending on a constant *oblivion* ($0 \leq oblivion \leq 1$) that expresses how quickly the acceleration reacts to changes in mining power. The computation of the next power uses the same formula as Bitcoin [25,9], adapted to our context: the ratio between the maximal (hence worse) deadline's value $2^{8 \cdot size(h_{deadline})}$ and the actual deadline's value expresses how much space has been used to compute the deadline.

Definition 19 (Next functions). *Let* $b \in \mathsf{Blocks}$ *and* $\delta \in \mathsf{Deadlines}$*. We define*

$$\begin{aligned} \tau_{next}(b,\delta) &= b.\tau + waitingTime(\delta, b.\alpha), \\ weightedBeat_{next}(b,\delta) &= \begin{array}{c} waitingTime(\delta, b.\alpha) \cdot oblivion \\ +b.weightedBeat \cdot (1 - oblivion) \end{array}, \\ \alpha_{next}(b,\delta) &= \frac{b.\alpha \cdot weightedBeat_{next}(b,\delta)}{beat}, \\ power_{next}(b,\delta) &= b.power + \frac{2^{8 \cdot size(h_{deadline})}}{be2nat(\delta.value) + 1}. \end{aligned}$$

Def. 20 shows how a next block is constructed, once its deadline has been chosen.

Definition 20 (Next block). *Let* $b \in \mathsf{Blocks}$ *and* $\delta \in \mathsf{Deadlines}$*. We define* $block_{next}(b,\delta) \in \mathsf{NonGenesisBlocks}$ *as*

$$block_{next}(b,\delta) = \left\langle \begin{array}{c} \tau_{next}(b,\delta), \alpha_{next}(b,\delta), power_{next}(b,\delta), \\ weightedBeat_{next}(b,\delta), \langle height(b)+1, \delta \rangle, h_{block}(b) \end{array} \right\rangle,$$

where $h_{block}(b)$ *is the application of* h_{block} *to the byte representation of* b*.*

A blockchain is a set of blocks, linked through their *previousBlockHash* field. It must contain exactly one genesis block; there is no hash collision among its blocks; and all its blocks must satisfy the *consensus rules*.

Definition 21 (Blockchain, Consensus). *A* blockchain *is a set* $B \subseteq \mathsf{Blocks}$ *such that:*

1. *there is exactly one* $b \in B \cap \mathsf{GenesisBlocks}$*, written as* $genesis(B)$*;*
2. *for each hash* h *of* h_{block}*, there is at most one* $b \in B$ *such that* $h_{block}(b) = h$*, written as* $block(B, h)$*;*
3. *for each* $b \in B$*, the predicate* $consensus(B, b)$ *holds, where*
   - *if* $b \in \mathsf{GenesisBlocks}$*, then* $consensus(B, b)$ *is just the consensus rule:*
     (a) *b is not created in the future:* $b.\tau \leq \tau_{now}$*;*
   - *if* $b \in \mathsf{NonGenesisBlocks}$*, then* $consensus(B, b)$ *is the logical conjunction of all the following consensus rules:*
     (a) *b is not created in the future:* $b.\tau \leq \tau_{now}$*;*
     (b) *the deadline of b (that is,* $b.trunk.\delta$*) is valid (Def. 13);*
     (c) *there are no dangling pointers:* $p = block(B, b.previousBlockHash)$ *exists;*
     (d) *b's deadline answers the challenge of p (Def. 18):* $challenge_{next}(p) = b.trunk.\delta.challenge$
     (e) *b is p's next block wrt. b's deadline (Def. 20):* $b = block_{next}(p, b.trunk.\delta)$.

The above consensus rules, reconstructed and interpolated from [22,21], do not constrain the prologs of the deadlines: each block can have an arbitrary prolog. Sec. 5 will show why it is useful to restrain prologs with extra consensus rules.

Definition 22 (Blockchain network). *A* blockchain network *is a network of* peers *(computers), each connected to the other peers, each holding its own vision of the blockchain, all for the same genesis block. Each peer holds a plot (Def. 6) on disk, starts with a blockchain that only holds the genesis block and runs, concurrently, the* block mining *algorithm and the* block mined *algorithm.*

Def. 22 simplifies the picture very much: it assumes that peers are fully connected, never disconnect and never need to synchronize. In practice, peers do not hold plots but rely on (one or more) external services (miners) that hold the plots. The goal here is to keep the picture as simple as possible and concentrate on the properties of Signum's consensus only: Def. 22 does not pretend to describe a real blockchain implementation in detail.

Def. 22 defines the block mining algorithm. It is an infinite loop that looks for the most powerful block p in blockchain (step 1), derives a challenge c from p (step 2), uses a plot to compute a best deadline δ for c (step 3), computes the next block b' for δ (step 4) and waits for δ to expire (step 5). Then it adds b' in blockchain (step 6), whispers b' to all peers (step 7) and restarts.

Algorithm 2 (Block mining) *The* block mining *algorithm of a peer P, holding blockchain B, is the following infinite loop:*

1. *identify a most powerful*[7] *block p in B;*
2. *compute* $c = challenge_{next}(p)$ *(Def. 18);*

[7] In theory, more *most powerful blocks* might exist in blockchain, although this is highly unlikely; in that case, step 1 will choose any of them.

3. *compute* $\delta' = \delta(plot, c)$ *(Def. 12), where plot if the plot of* P*;*
4. *compute* $b' = block_{next}(p, \delta')$ *(Def. 20);*
5. *wait until* $b'.\tau \leq \tau_{now}$*;*
6. *add* b' *to* B*;*
7. *whisper* b' *to the peers connected to* P*;*
8. *go back to step 1.*

The block mined algorithm receives a block whispered from some connected peer (step 1), checks its validity (step 2) and adds it to the blockchain.

Algorithm 3 (Block mined) *The* block mined *algorithm of a peer* P*, holding blockchain* B*, is the following infinite loop:*

1. *wait for a block* b *whispered from some connected peer* P'*;*
2. *if* $B \cup \{b\}$ *is a blockchain, add* b *to* B*;*
3. *go back to step 1.*

In practice, step 2 of Alg. 3 could only allow the addition of b if it looks *powerful enough*, in order to avoid keeping useless blocks. This is not relevant here. Moreover, if the whispered block b at step 2 of Alg. 3 is more powerful than b' at step 5 of Alg. 2, a rational peer would interrupt waiting at step 5 of Alg. 2, discard b' and restart Alg. 2 from step 1, since the whispered b is better than the block b' that it is being mined, hence it is wiser to stop waiting and start mining on top of b. These are optimizations and are not considered here.

Peers check the validity of blocks coming from outside (step 2 of Alg. 3), since they do not trust their connected peers. Instead, they do not check the validity of the blocks that they mine themselves (step 6 of Alg. 2), since they are valid by construction. Namely, Prop. 1 guarantees that B remains a blockchain in every peer. The hypothesis on no hash collision is standard. Namely, all existing blockchains make that hypothesis and they would all collapse otherwise, because they all identify blocks by their hash.

Proposition 1. *If no hash collision occurs at step 6 of Alg. 2, then the set* B *of blocks in each peer is a blockchain.*

5 Prolog Structure, Protection against Attacks

This section uses the formalization of the previous sections to extend the framework in [22] and understand if nothing-at-stake attacks might work for Signum.

The peers of Def. 22 have two functions: to find new deadlines and to package new blocks with such deadlines. In practice, two machines perform each of them: one (the actual miner) finds deadlines by using plots: it must have a large disk space; the other (the actual peer) receives deadlines, packages and whispers new blocks: it must have a good network connection. Such machines will work for a blockchain network with a given chain identifier [2]. Therefore, we propose to use, as prologs, the byte representation of the following triple:

$$\langle chainIdentifier, publicKeyOfPeer, publicKeyOfMiner \rangle$$

(in [22], prologs are just *publicKeyOfMiner*). Moreover, we add two new consensus rules to Def. 21. Informally, one requires that the chain identifier of the blockchain is $b.trunk.\delta.\pi.chainIdentifier$; and the other requires that the signature of b is verified with $b.trunk.\delta.\pi.publicKeyOfPeer$. This has many advantages:

- the public keys of peer and miner can be used to remunerate them for their contribution, in an application-specific way;
- the creator of a plot must specify the public key of the peer: hence that plot can only be used to create deadlines for that given peer and miners become dedicated to that given peer, instead of working, with the same plot, for many peers. This creates a sort of miner fidelization and allows peers to compete by offering different remuneration schemes to miners;
- the creator of a plot must specify the chain identifier of the blockchain. Therefore, it becomes impossible to use the same plot for mining two blockchain networks at the same time. This protects against newborn attacks.

Note that this structure for the prologs entails that deadlines are around 170 bytes (assuming hashes, signatures and chain identifier to be 32 bytes each).

In [16], Burstcoin was criticized because the validation of a deadline (Def. 13) requires to run Alg. 1 that, they say, requires hashing $8 \cdot 10^6 \cdot 32 = 256000000$ bytes. It must be stated that such hashing can actually be computed in a few milliseconds nowadays, with a minimal energy cost. Moreover, it is largely dominated by that for verifying the transactions in a block, in particular for Signum that allows smart contracts. Therefore, this time for deadline validation seems largely acceptable to us. In any case, step 2 of Alg. 1 currently uses a threshold κ, that was possibly missing when [16] was written. Considering κ and the specific values and hashing used in [22] (rightmost column of Tab. 1), step 2 hashes at most κ bytes and is iterated $2 \cdot \#scoops$ times, that is, it hashes at most 33554432 bytes. Step 3 hashes $32 \cdot 2 \cdot 4096$ bytes plus the size of *seed*, which is reasonably at most 200 bytes. That is, it hashes 262344 bytes. In total, Alg. 1 hashes 33816776 bytes, around 8 times less than what was reported in [16].

In Appendix B of [16], a potential block-grinding attack was hinted for Burstcoin, since, in [22], the next challenge of a previous block used its undefined *previous block generator*: "this seems possible to be grinded, by trying different sets of transactions to include in a block" [16]. As discussed just after our Def. 17, that notion is just the prolog $trunk.\delta.\pi$ of the deadline of the trunk in the previous block. Consequently, it cannot be grinded. That is, Signum is protected against block-grinding attacks. But we sympathize with the authors of [16]: without a formalization, it was impossible to exclude that attack.

We furthermore observe that Signum has some protection against challenge-grinding attacks. The challenge for the a block is actually uniquely determined by the trunk of its previous block (rule (d) of Def. 21 and Def. 18) but a peer might select suboptimal deadlines at step 3 of Alg. 2 (*ie.*, not the minimal one of Def. 12), hoping that such a sacrifice will lead to subsequent challenges for which it can find very good deadlines. But Prop. 2 shows that that choice does not pay off: deadlines do not affect the sequence of challenges, only the plots do.

Proposition 2. *Let B be a blockchain, $plot \in \mathsf{Plots}$, $b_0, \ldots, b_n$ and $b'_0, \ldots, b'_n$ be two sequences of blocks in B, rooted at the same $b_0 = b'_0$, mined by using plot only (possibly with suboptimal choices of the deadlines), that is,*

$$b_i.trunk.\delta = \delta(nonce_i, plot.\pi, challenge_{next}(b_{i-1}))$$
$$b'_i.trunk.\delta = \delta(nonce'_i, plot.\pi, challenge_{next}(b'_{i-1}))$$

for suitable $nonce_i, nonce'_i \in plot.nonces$, for every $1 \leq i \leq n$. Then it holds $challenge_{next}(b_j) = challenge_{next}(b'_j)$ for every $0 \leq j \leq n$.

Intuitively, the sequence of challenges depends only on the sequence of prologs used for mining, that is always the same by using a given plot. Therefore, a challenge grinding attack in Signum would require one to use more plots, with prologs with different keys, all in control of the peer, and grind among the plots. However, the number of plots is limited by the space allocated to the algorithm and the same set of plots would end up being used at each mining step, since they are too expensive to generate on the fly during grinding. This highly limits the benefits of grinding. For additional protection, it is always possible to use one of the techniques presented in [16] (see Sec. 6).

Signum does not seem to have any protection against mining on different chains instead. The only possibility here seems to use the penalty transactions used in [16] and challenges that are not built from the trunk of the previous block but from that of a predecessor block deeper in the chain (see Sec. 6).

Appendix B of [16] shows that the original definition of Burstcoin had a time/memory tradeoff: it was possible to store only each h_{final} instead of each full nonce (see step. 3 of Alg. 1) and reconstruct (some) scoops on demand. The resulting (mining power)/(space used) ratio was proportionally higher than by storing the full nonces, "at the price of having to compute a modest number of extra hashes" [16]. We agree with the attack but not with the *modest number*: the discussion in [16] confuses plots with nonces and does not recognize that that modest number must be computed for all nonces in the plot, which are easily too many, considering the cost of the shabal256 hashing algorithm (Tab. 1). In any case, the developers of Signum have modified Alg. 1 with the addition of step 5, in order to cope with this time/memory tradeoff. That extra step requires one to compute all h_i's in order to compute the scoops. Therefore, this specific time/memory tradeoff does not occur anymore. Of course, proving that the addition of step 5 is enough to ban *all* time/memory tradeoffs remains an open question, a daunting task that goes well beyond the scope of this paper.

6 Related Work

Proof of work was originally meant as protection against email spam [7]: senders must perform some work to have their emails accepted by recipients. Ethereum started with proof of work [3] and later moved to proof of stake. The latter can be seen as a Byzantine consensus algorithm, as pioneered by Tendermint [12]. Most current blockchains use some form of proof of stake nowadays.

Traditional properties of consensus algorithms are *consistency* (all honest peers eventually converge to the same blockchain), *liveness* (a transaction submitted to the network gets included in blockchain after a *reasonable* time) and *order-fairness* (peers include transactions in blockchain in their arrival order). This paper does not discuss them because Signum's consensus behaves exactly as Bitcoin's proof of work *wrt.* these properties. Namely, step 1 of Alg. 2 guarantees that honest peers select the same chain (if whispering is working); while liveness and order-fairness hold for honest peers but not if a dishonest peer holds a large part of the total mining power (as in the 51% attack). Note that order-fairness is more easily lost in other consensus algorithms, such as Byzantine consensus algorithms, where validators can decide the history independently from their mining power. Solutions, in that context, are reported in [11].

The theory of proof of space was independently developed in two seminal papers [4,8], both based on directed acyclic graphs (DAGs) of high pebbling complexity. Pebbling, here, is a directed hash decoration of the nodes of the DAG, as in a Merkle tree. A prover must hold such a (big) DAG and its pebbling on disk, in order to answer, efficiently, challenges with proofs that should convince a verifier that the prover is actually holding data on disk. While [8] requires space to remain allocated between challenges (proof of *persistent* space), [4] requires one to allocate space only when answering challenges (proof of *transient* space or *proof of secure erasure*, as [8] calls it). Both solutions have an initialization phase, when the verifier performs a deep challenge of the prover and stores the resulting (big) proof in blockchain, followed by an execution phase, when the verifier challenges the prover for each new block. Also [18] uses pebbling for stacked expander graphs, to get simpler, more efficient, provably space-hard solutions. It works for both proof of transient space and proof of persistent space. It includes a nice review on proof of space and related techniques: memory-hard functions, proof of secure erasure, provable data possession, proof of retrievability.

Time/memory tradeoffs are studied in [19]. They occur if a prover can store only a part of the data on disk, with a less than proportional degradation of mining power. A good proof of space algorithm makes it difficult to recover the missing part when answering challenges. For graph pebbling, the initialization phase prevents most cheating (that is, keeping incomplete data on disk). In [19], the size of the portion of the file not kept on disk is related to the consequent time complexity degradation for computing the missing part. Ideally, the full file and pebbling must be kept on disk for having no time complexity explosion, but they show that this is not the case in existing solutions and provide sufficient conditions for the initialization phase, that guarantee the ideal result.

The use of graph pebbling seems to dominate the literature on proof of space, but [1] proposes an alternative theory, that supports the Chia network [6,5]: a proof of sequential work on top of a proof of space, based on challenges about the inversion of a random function, for which time/memory tradeoffs have been solved. However, this is not a pure proof of space. Another alternative is the proof of retrievability in [10]: the verifier sends, initially, a large file to the prover (miner) that later challenges, repeatedly, to see if it still keeps the file on disk.

Its apparent simplicity is jeopardized by the difficulty of sharing big files among *all* (present and future) peers, for *all* (past, present and future) miners.

We are only aware of one implementation of graph-pebbling proof of space: SpaceMint [16], previously Spacecoin [17]. It is not a blockchain but a discontinued prototype of only the consensus protocol of [8]. Nevertheless, [16] exposes problems and solutions related to the actual game theory and implementation of proof of space. For instance, it uses the verifier's public key as an input parameter for pebbling, to discourage the use of mining pools, often seen negatively [14]. Furthermore, it provides solutions for nothing-at-stake problems. Against block grinding, it makes challenges independent from the transactions in the blocks, by splitting the blockchain in a proofs blockchain and in a transactions blockchain: only the first is used for mining, and the two are connected with the miner's signature. Against challenge grinding, it lets past blocks influence the quality of short sequences of future blocks only. A similar but more drastic solution in [6] uses the same challenge for several consecutive blocks, since it is unlikely that it will be good for all of them. Against mining on multiple chains, [16] proposes to spot such behavior and impose a penalty transaction to the culprit. Experiments in [16] include an estimation of the size of Spacemint's proofs: in the initialization phase, they are between two and three megabytes; in the execution phase, they can be optimized to around 100 kilobytes. Proofs must be persistently stored in blockchain (for each new miner, in the first case, and for each new block, in the second), which makes the blockchain's size much larger than in Bitcoin, and requires miners to hold cryptocurrency even before starting mining.

Signum [20], previously Burstcoin, has been launched in 2014, with possibly the first ever language for smart contracts, and is still active. Appendix B of [16] reports a formalization of an old version of Alg. 1.

Newborn attacks are considered in [24]. Their solution is to split the space for mining on many chains, with an incentive to allocate, for each chain, a space proportional to the market value of that chain.

7 Conclusion

In the context of proof of space consensus, Signum's advantage is its simplicity, the small size of its proofs (deadlines, around 200 bytes) and the absence of an initialization phase and transactions. Moreover, it is the only fully implemented and deployed solution and supports smart contracts. Its drawback is that it is theoretically possible to mine new blocks in a proof of work style, although this is not been observed in practice up to now. Our formalization of Signum's consensus is valuable because it sheds light on a blockchain network that runs since ten years but was missing any formal definition. Moreover, it allowed us to understand that Signum is free from block-grinding attacks and is largely protected from challenge-grinding attacks.

References

1. H. Abusalah, J. Alwen, B. Cohen, D. Khilko, K. Pietrzak, and L. Reyzin. Beyond Hellman's Time-Memory Trade-Offs with Applications to Proofs of Space. In T. Takagi and T. Peyrin, editors, *Proc. of Advances in Cryptology, the 23rd International Conference on the Theory and Applications of Cryptology and Information Security (ASIACRYPT'17), part II*, volume 10625 of *Lecture Notes in Computer Science*, pages 357–379, Hong Kong, China, December 2017. Springer.
2. A. M. Antonopoulos. *Mastering Bitcoin: Programming the Open Blockchain*. Oreilly & Associates Inc, 2nd edition, June 2017.
3. A. M. Antonopoulos and G. Wood. *Mastering Ethereum: Building Smart Contracts and Dapps*. Oreilly & Associates Inc, 1st edition, November 2018.
4. G. Ateniese, I. Bonacina, A. Faonio, and N. Galesi. Proofs of Space: When Space Is of the Essence. In M. Abdalla and R. De Prisco, editors, *Proc. of the 9th International Conference on Security and Cryptography for Networks (SCN'14)*, volume 8642 of *Lecture Notes in Computer Science*, pages 538–557, Amalfi, Italy, September 2014. Springer.
5. Chia Network. `https://www.chia.net`. Accessed on August 26, 2024.
6. B. Cohen and K. Pietrzak. The Chia Network Blockchain. `https://www.chia.net/wp-content/uploads/2022/07/ChiaGreenPaper.pdf`, 2019. Accessed on September 2, 2024.
7. C. Dwork and M. Naor. Pricing via Processing or Combatting Junk Mail. In E. F. Brickell, editor, *Proc. of the 12th Annual International Cryptology Conference on Advances in Cryptology (CRYPTO'92)*, volume 740 of *Lecture Notes in Computer Science*, pages 139–147, Santa Barbara, California, USA, August 1992. Springer.
8. S. Dziembowski, S. Faust, V. Kolmogorov, and K. Pietrzak. Proofs of Space. In R. Gennaro and M. Robshaw, editors, *Proc. of Advances in Cryptology (CRYPTO 2015) - 35th Annual Cryptology Conference, part II*, volume 9216 of *Lecture Notes in Computer Science*, pages 585–605, Santa Barbara, CA, USA, August 2015. Springer.
9. J. Garay, A. Kiayias, and N. Leonardos. The Bitcoin Backbone Protocol with Chains of Variable Difficulty. In J. Katz and H. Shacham, editors, *Proc. of the 37th International Cryptology Conference (CRYPTO'17), Part I*, volume 10401 of *Lecture Notes in Computer Science*, pages 291–323, Santa Barbara, CA, USA, August 2017. Springer.
10. A. Juels and B. S. Jr. Kaliski. Pors: Proofs of Retrievability for Large Files. In P. Ning, S. De Capitani di Vimercati, and Syverson. P. F., editors, *Proc. of the 2007 ACM Conference on Computer and Communications Security (CCS'07)*, pages 584–597, Alexandria, Virginia, USA, October 2007. ACM.
11. M. Kelkar, F. Zhang, S. Goldfeder, and A. Juels. Order-Fairness for Byzantine Consensus. In D. Micciancio and T. Ristenpart, editors, *Proc. of the 40th International Cryptology Conference (CRYPTO'20)*, volume 12172 of *Lecture Notes in Computer Science*, pages 451–480, Santa Barbara, CA, USA, August 2020. Springer.
12. J. Kwon. Tendermint: Consensus without Mining. `https://www.weusecoins.com/assets/pdf/library/Tendermint%20Consensus%20without%20Mining.pdf`, 2014. Accessed on August 26, 2024.
13. A. Miller, A. Juels, E. Shi, B. Parno, and J. Katz. Permacoin: Repurposing Bitcoin Work for Data Preservation. In *Proc. of the IEEE Symposium on Security and Privacy (SP'14)*, pages 475–490, Berkeley, CA, USA, May 2014. IEEE Computer Society.

14. A. Miller, A. E. Kosba, J. Katz, and E. Shi. Nonoutsourceable Scratch-Off Puzzles to Discourage Bitcoin Mining Coalitions. In I. Ray, N. Li, and C. Kruegel, editors, *Proc. of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 680–691, Denver, CO, USA, October 2015. ACM.
15. S. Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System. `https://bitcoin.org/bitcoin.pdf`, October 2008. Accessed on August 26, 2024.
16. S. Park, A. Kwon, G. Fuchsbauer, P. Gazi, J. Alwen, and K. Pietrzak. SpaceMint: A Cryptocurrency Based on Proofs of Space. In S. Meiklejohn and K. Sako, editors, *Proc. of the 22nd International Conference on Financial Cryptography and Data Security (FC'18), Revised Selected Papers*, volume 10957 of *Lecture Notes in Computer Science*, pages 480–499, Nieuwpoort, Curaçao, February - March 2018. Springer.
17. S. Park, K. Pietrzak, J. Alwen, G. Fuchsbauer, and P. Gazi. Spacecoin: A Cryptocurrency Based on Proofs of Space. IACR Cryptology ePrint Archive, `https://eprint.iacr.org/2015/528.pdf`, 2015. Accessed on August 26, 2024.
18. L. Ren and S. Devadas. Proof of Space from Stacked Expanders. In M. Hirt and A. D. Smith, editors, *Proc. of the 14th International Conference on Theory of Cryptography (TCC'16-B), part I*, volume 9985 of *Lecture Notes in Computer Science*, pages 262–285, Beijing, China, October - November 2016.
19. L. Reyzin. Proofs of Space with Maximal Hardness. *IACR Cryptol. ePrint Arch.*, page 1530, 2023.
20. Signum Community Website and Documentation Project. `https://wiki.signum.network`. Accessed on August 26, 2024.
21. Signum Node Source Code. `https://github.com/signum-network/signum-node`. Accessed on August 27, 2024.
22. Signum Plotting and Mining Technical Information. `https://wiki.signum.network/signum-plotting-technical-information/index.htm`. Accessed on August 26, 2024.
23. SpaceMint Source Code. `https://github.com/kwonalbert/spacemint`. Accessed on August 26, 2024.
24. S. Tang, J. Zheng, Y. Deng, Z. Wang, Z. Liu, D. Gu, Z. Liu, and Y. Long. Towards a Multi-chain Future of Proof-of-Space. In S. Chen, K.-K. R. Choo, X. Fu, W. Lou, and A. Mohaisen, editors, *Proc. of the 15th EAI International Conference on Security and Privacy in Communication Networks (SecureComm'19), part I*, volume 304 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pages 23–38, Orlando, FL, USA, October 2019. Springer.
25. G. Walker. Longest Chain – The Chain of Blocks Nodes Adopt as Their Blockchain. `https://learnmeabitcoin.com/technical/blockchain/longest-chain/`, 2024. Accessed on September 12, 2024.

A Proofs

Proof of Prop. 1 at page 11
Let us prove it by induction on the number of blocks added to B in Algs. 2 and 3. The thesis is true when the peer starts, by Def. 22. Alg. 3 keeps B as a blockchain, since it explicitly checks it (step 2 of Alg. 3). It remains to show that, if B is a blockchain at the beginning of step 6 of Alg. 2, then $B \cup \{b'\}$ is a blockchain at its end. By Def. 20, it is $b' \in \mathsf{NonGenesisBlocks}$. Therefore, property 1 of Def. 21 remains true. By the assumption on no hash collision, property 2 remains true. It remains to prove that property 3 remains true as well. Let $b \in B \cup \{b'\}$.

- If $b \in \mathsf{GenesisBlocks}$, it is $b \neq b'$ and then $b \in B$; since τ_{now} cannot decrease with the time, by inductive hypothesis the property $b.\tau \leq \tau_{now}$ was true at the start of step 6 of Alg. 2 and is still true at its end.
- If $b \in \mathsf{NonGenesisBlocks}$ and $b \in B$, by inductive hypothesis all consensus rules held when b was added to B and they must still hold now, since τ_{now} only increases with the time and since no block is ever removed from B (nor replaced in B, since there is no hash collision).
- If $b \in \mathsf{NonGenesisBlocks}$ and $b \notin B$, it must be $b = b'$. Let us prove that each consensus rule in Def. 21 holds.
  (a) This rule holds by step 5 of Alg. 2 and by the fact that τ_{now} cannot decrease from step 5 to step 6;
  (b) It is

$$\begin{aligned} b.trunk.\delta &= b'.trunk.\delta \\ \text{(step 4 of Alg. 2 and Def. 20)} &= \delta' \\ \text{(step 3 of Alg. 2)} &= \delta(plot, c) \\ \text{(Def. 12)} &= \delta(nonce, plot.\pi, c) \end{aligned}$$

  for some $nonce \in plot.nonces$. The deadline $\Delta = \delta(nonce, plot.\pi, c)$ is valid (Def. 13) since, by Def. 11:

$$\begin{aligned} \Delta.value &= value(nonce, c) \\ &= value(nonce, \Delta.challenge) \\ \text{(Def. 6)} &= value(nonce(nonce.p, plot.\pi), \Delta.challenge) \\ \text{(Def. 11)} &= value(nonce(\Delta.p, \Delta.\pi), \Delta.challenge). \end{aligned}$$

  (c) It is

$$\begin{aligned} b.previousBlockHash &= b'.previousBlockHash \\ \text{(step. 4 of Alg. 2)} &= block_{next}(p, \delta').previousBlockHash \\ &= h_{block}(p), \end{aligned}$$

  where p is the most powerful block in B (by step. 1 of Alg. 2). Therefore, $block(B, b.previousBlockHash)$ exists and coincides with the block p selected at step 1 of Alg. 2. This fact is used in the subsequent points of this proof.

(d) By step 2 of Alg. 2, it is

$$\begin{aligned} challenge_{next}(p) &= c \\ (\text{Defs. 12 and 11}) &= \delta(plot, c).challenge \\ (\text{step 3 of Alg. 2}) &= \delta'.challenge \\ (\text{step 4 of Alg. 2 and Def. 20}) &= b'.trunk.\delta.challenge \\ &= b.trunk.\delta.challenge. \end{aligned}$$

(e) It is

$$\begin{aligned} b &= b' \\ (\text{step. 4 of Alg. 2}) &= block_{next}(p, \delta') \\ (\text{Def. 20}) &= block_{next}(p, b.trunk.\delta). \end{aligned}$$

□

Proof of Prop. 2 at page 13

Let us proceed by induction on n. If $n = 0$, the thesis holds since $b_0 = b'_0$. Let $n > 0$ and assume that the thesis holds for $n-1$. Let us prove it for n. By inductive hypothesis, it is enough to prove that $challenge_{next}(b_n) = challenge_{next}(b'_n)$. Since $n > 0$, by consensus rule (e) of Def. 21 and by Def. 20, it is $b_n \in \mathsf{NonGenesisBlocks}$. Therefore, by Def. 18, it is

$$\begin{aligned} challenge_{next}(b_n) &= challenge_{next}(b_n.trunk) \\ (Def.\ 17) &= \langle be2nat(h_{generation}(\sigma \bowtie nat2be(b_n.trunk.height + 1))) \\ &\quad \bmod \#scoops, \sigma\rangle \\ \begin{pmatrix} \text{consensus} \\ \text{rule (e)} \\ \text{of Def. 21,} \\ \text{and Def. 20} \end{pmatrix} &= \langle be2nat(h_{generation}(\sigma \bowtie nat2be(b_0.trunk.height + n + 1))) \\ &\quad \bmod \#scoops, \sigma\rangle \end{aligned}$$

and similarly

$$\begin{aligned} challenge_{next}(b'_n) = \langle &be2nat(h_{generation}(\sigma' \bowtie nat2be(b_0.trunk.height + n + 1))) \\ &\bmod \#scoops, \sigma'\rangle \end{aligned}$$

where

$$
\begin{aligned}
\sigma &= h_{generation}(b_n.trunk.\delta.challenge.\sigma \bowtie b_n.trunk.\delta.\pi) \\
\begin{pmatrix} b_n \text{ has been mined} \\ \text{by using } plot, \\ \text{and Def. 11} \end{pmatrix} &= h_{generation}(b_n.trunk.\delta.challenge.\sigma \bowtie plot.\pi) \\
\begin{pmatrix} \text{consensus rule (d)} \\ \text{of Def. 21} \end{pmatrix} &= h_{generation}(challenge_{next}(b_{n-1}).\sigma \bowtie plot.\pi) \\
\begin{pmatrix} \text{inductive} \\ \text{hypothesis} \end{pmatrix} &= h_{generation}(challenge_{next}(b'_{n-1}).\sigma \bowtie plot.\pi) \\
\begin{pmatrix} \text{consensus rule (d)} \\ \text{of Def. 21} \end{pmatrix} &= h_{generation}(b'_n.trunk.\delta.challenge.\sigma \bowtie plot.\pi) \\
\begin{pmatrix} b'_n \text{ has been mined} \\ \text{by using } plot, \\ \text{and Def. 11} \end{pmatrix} &= h_{generation}(b'_n.trunk.\delta.challenge.\sigma \bowtie b'_n.trunk.\delta.\pi) \\
&= \sigma'.
\end{aligned}
$$

It follows that $challenge_{next}(b_n) = challenge_{next}(b'_n)$. □