Foundations of Blockchains
Lectures #9: Permissionless Consensus and Proof-of-Work
(IN PROGRESS)*

Tim Roughgarden†

1 The Upshot

1. A permissionless protocol operates without knowledge of the set of nodes running the protocol.

2. A permissionless subroutine for randomly sampling one of the nodes running the protocol is the key to permissionless versions of BFT-type and longest-chain consensus protocols.

3. Permissionless protocol design is hard because a Byzantine node can costlessly execute Sybil attacks—generating many public-private key pairs and masquerading as multiple nodes.

4. A consensus protocol (like BFT-type or longest-chain) determines the state of the blockchain given nodes' proposals and votes for blocks. A sybil-resistance mechanism (like proof-of-work or proof-of-stake) determines which nodes can make block proposals or vote.

5. Nakamoto consensus refers to the pairing of longest-chain consensus with proof-of-work sybil-resistance.

6. Lecture 8 shows that the only ingredient missing from a permissionless version of longest-chain consensus with provable consistency and liveness guarantees is a permissionless node selection subroutine that selects honest nodes more frequently than Byzantine ones.

†Email: tim.roughgarden@gmail.com. The preparation of these notes was partially supported by a seed grant from the Columbia-IBM Center for Blockchain and Data Transparency. These notes are a work in progress; corrections and feedback are welcome.

7. In proof-of-work, the selected node is the first node to solve a hard puzzle.

8. The canonical hard puzzle is to find an input to a cryptographic hash function (such as SHA-256) that hashes to a sufficiently small number.

9. The random oracle assumption asserts that a cryptographic hash function is for all practical purposes indistinguishable from a perfectly random function.

10. In Nakamoto consensus, candidate puzzle solutions must encode a single block proposal and a predecessor for that block.

11. In proof-of-work, under the random oracle assumption, each node is selected with probability proportional to the overall amount of computational power (or "hashrate") that it contributes to the protocol, independent of how many identities it might use.

12. Under the random oracle assumption, Nakamoto consensus achieves consistency and liveness (with high probability) in the super-synchronous model provided less than half of the total hashrate is controlled by Byzantine nodes.

13. The guarantees of Nakamoto consensus do not contradict the PSL-FLM impossibility result from Lecture 3 because proof-of-work sybil-resistance makes the simulation of honest nodes costly.

2 Permissionless Consensus

Recap of the permissioned setting. Thus far we've been studying consensus protocols in the safe confines of the *permissioned* setting, in which the nodes running the protocol are fixed and known in advance. This scenario made perfect sense when computer science researchers started developing a theory of consensus protocols in the 1980s, when a typical application might be something like database replication (with a single company buying a bunch of nodes to each store a copy of the database in the interests of very high uptime).

Thus far, we're also been making a PKI (public key infrastructure) assumption. That is, we're assuming that secure signature schemes exist, that each node has its own public-private key pair, and the all nodes' public keys are known to all nodes at the beginning of the protocol (with each node signing its messages and verifying signatures on all received messages). This is a "trusted setup" assumption, meaning we just assume that all nodes' public keys were somehow shared correctly in advance of the protocol's commencement (just as we assumed that all nodes somehow obtained a correct version of the protocol's code). This trusted setup assumption is also totally reasonable in our running example of database replication by a centralized organization. Tendermint (Lecture 7) is a canonical example of a permissioned consensus protocol.[1]

[1] Blockchains that rely on a permissioned consensus protocol are sometimes called "private blockchains" or "proof-of-authority blockchains."

Open participation and permissionless protocols. The most famous blockchain protocols, such as Bitcoin and Ethereum, do not operate in the permissioned setting. To viscerally appreciate this point, I encourage you to take a break from this lecture and spin up on your laptop or desktop a full node that validates the Bitcoin or Ethereum blockchains. These blockchains have been running for years and have no idea who you are, and yet you can simply download software, sync your machine with the blockchain-so-far, and join the party. This is obviously very different from the classical permissioned setting.

Remember in Lecture 1 when I described a mental model of a blockchain as a (virtual) computer in the sky, with no owner or operator? Really what this meant was that this computer has *lots* of operators (coordinated via a consensus protocol), and moreover you yourself can become one of these operators, if you wish. This is the vision of permissionless consensus, and it is perhaps the most radical and audacious conceit in the original Bitcoin protocol.

The challenges of permissionless consensus. A permissionless consensus protocol has an unknown and possibly ever-changing set of nodes running it, which would seem to throw an immediate wrench into, for example, a Tendermint-style protocol. In Tendermint, with 100 nodes say, nodes wait to collect and see evidence of 67 votes before proceeding with any action. If you don't know how many nodes there are, how do you know how many votes to collect? Similarly, without knowing which nodes are running the protocol, how would you ever implement the round-robin approach to leader selection (in Tendermint, or in longest-chain consensus)?

It also makes sense to think about permissionless consensus protocols as operating via broadcast channels—perhaps implemented via a gossip protocol in a free-to-join peer-to-peer network—rather than point-to-point communication.[2] So, we'll imagine that whenever an honest node sends a message, it automatically broadcasts that messages to all other nodes. (We don't want to impose any unjustified restrictions on what Byzantine nodes can do, so we'll continue to allow them to, for example, send conflicting messages to different sets of honest nodes.) Similarly, clients (in the sense of a state machine replication protocol) are assumed to submit their transactions via a broadcast channel that is listened in on by all the nodes that are currently running the protocol.[3] This "broadcast-only" communication restriction shouldn't bother you too much—if you go back over our pseudocode for the Tendermint protocol in Lecture 7, for example, you'll see that the nodes are basically already communicating via broadcast messages, anyway (e.g., announcing or echoing quorum certificates).[4]

[2] How do these peer-to-peer gossip protocols work? Good question, and there are interesting answers (and also open research questions), but as stated in Lecture 1 such "layer 0" questions are outside the scope of this lecture series. To read more about this topic, start with [**?**].

[3] Note the creep in our trusted setup assumption: we need to assume nodes (including newly joining ones) know how to access to the correct broadcast channel/peer-to-peer network, and that clients know the correct channel/network to submit their transaction to.

[4] More communication-efficient versions of Tendermint and other BFT-type protocols do take advantage of point-to-point communication, but that's not our concern here.

The dream of permissionless consensus is an ambitious one, and in light of the seeming incompatibilities between the constraints of the permissionless setting and the consensus protocols that we've seen thus far, you'd be right to question permissionless consensus protocols with provable guarantees could even exist!

Such protocols *do* exist—if they didn't, there'd be no lecture series!—but we'll need some additional ideas. As mentioned, an immediate obstacle to implementing a BFT-type protocol in a permissionless setting is the seeming impossibility of supermajority voting when the set of voters is unknown. A second issue, relevant to both BFT-type protocols and longest-chain consensus, it how to select leaders (i.e., which node gets to propose the next block). "Round-robin order" doesn't appear to make sense with an unknown and ever-changing set of nodes. "Uniformly random selection" also doesn't seem to make sense—if there were 100 nodes, in each round a give node should be selected as a leader with 1% probability. If there were 1000 nodes, it should be a 0.1% probability. But if the protocol doesn't know how many nodes there are, how can it know the appropriate probability with which to select a node?

3 Random Sampling and Sybil Attacks

3.1 Why Permissionless Random Sampling Is Useful

The key idea for transforming both BFT-type and longest-chain consensus protocols into permissionless protocols is a permissionless and easily verifiable method of *random sampling* one of the nodes that's currently running the consensus protocol. Two skeptical (but related) questions should come immediately to mind: (i) sample from which distribution over nodes, exactly? (ii) in any case, how would you do that without knowing which nodes are running the protocol? Let's put these questions aside briefly, and outline how such a subroutine would allow us to translate our permissioned protocols to the permissionless setting.

BFT-type consensus. Consider the Tendermint protocol, for example. Recall from Lecture 7 that, in each round of the protocol, a single leader node proposes a (new or inherited) block and then the entire set of nodes casts and tracks (two stages of) votes for that round's block proposal. The idea is to use, in each round of the protocol, the assumed method of random sampling a node once to select that round's leader as well as $s-1$ additional times to select the rest of the "committee" of s nodes (here s is a parameter of the protocol, like $s = 20$ or $s = 100$, and is independent of the actual and unknown number n of nodes running the protocol). The s randomly selected nodes are then the nodes tasked with carrying out the current round of the Tendermint protocol. (If a node is randomly selected more than once for the committee, then it gets one distinct vote for each time it was selected.)[5] In

[5]There are a number of details that need to be worked out in any practical implementation of this idea, some of which we'll discuss further in Lecture 12. For example, how big should the committee size be? Bigger committees mean less chance of unluckily sampling a much-bigger-than-expected number of Byzantine node but more work and communication for the nodes. Another question is now frequently to resample the committee—one extreme is to resample every round (or even every stage of every round), the

effect, this approach uses the assumed random sampling method as a "wrapper" to reduce permissionless consensus to permissioned consensus.

Longest-chain consensus. The utility of a random sampling method is even more obvious in the case of longest-chain consensus, where each block is proposed unilaterally by a round's leader and nobody votes (other than the implicit votes in each leader's decision of which previous block to extend). Here, there's no need to select a committee. The assumed random sampling method can be invoked once in each round to select that round's leader (corresponding to step (2a) of the pseudocode in Lecture 8), who then unilaterally proposes blocks and and their predecessors (step (2b)).

3.2 Challenge: Sybils

OK, fine, but should a permissionless and verifiable method of random sampling even exist? For example, how can you select a node uniformly at random without knowing the set of possible candidates?

As an initial (bad) idea, imagine that we tried implement uniformly random node sampling as follows: (i) nodes register their public keys in a smart contract stored on the blockchain; (ii) whenever needed, the protocol selects one of the currently registered keys uniformly at random.[6] For example, if the method is used to select a block proposer for a round, then the other nodes know to ignore any block proposals for that round that are not signed with the private key corresponding to the selected registered public key.

The glaring issue behind this approach is that, while it's all fine and good to maintain a list of public keys, the protocol still has no idea about the mapping from registered public keys to actual physical nodes. A single Byzantine node could costlessly create thousands or millions of public-private key pairs (just type `ssh-keygen` at a Unix prompt) and register them all, in effect masquerading as a huge number of distinct nodes.[7] Then, if there were (say) 99 honest nodes that each registered only once, the one Byzantine node would be selected by the protocol as the block proposer almost every round!

In general, in computer science when someone talks about a Sybil attack they usually mean the manipulation of some protocol, through the creation of lots of identities. Basically, a single party masquerading as many. Sybil attacks show up in lots of parts of computer science, not just in blockchains but I hope it's very obvious why Sybil attacks are a real danger for any random sampling approach to permissionless consensus.

---

other extreme is to resample rarely (like every two weeks). More frequent resampling means more resilience to adversaries (e.g., harder to mount denial-of-service attacks on the chosen nodes) and also more complexity for the protocol and the nodes running it.

[6] Technically, the protocol would select one of the registered keys pseudorandomly, for example using the lower-order bits of the hash of some seed. (See Section **??** for a related discussion.) If nodes running the protocol know what hash function and seed the protocol is using, then they will also automatically know which of the registered public keys was selected.

[7] By contrast, you can think of the PKI assumption in a permissioned setting as asserting that there is a one-to-one correspondence between the public keys registered with the protocol and the nodes that are actually running the protocol.

In general, deliberately creating multiple identities in a system is often called a *Sybil attack*.[8] Systems in which Sybil attacks do not help the attacker are called *sybilproof* or *Sybil-resistant*. The naive approach to random sampling above is not Sybil-resistant. This does not, of course, automatically mean that all approaches to random sampling are likewise doomed to fail. Could there be a Sybil-resistant method for randomly selecting one of the nodes running a consensus protocol—a method where the probability of a node's selection is independent of how many identities they might hide behind?

The answer, while not at all obvious, turns out to be "yes." There are multiple methods for Sybil-resistant random sampling, and in this lecture series we'll focus on the two that are most widely used as of 2022:

Dominant Approaches to Sybil-Resistant Random Sampling

1. *Proof-of-work.* (See Section 5 for details.) This approach samples a node running the protocol with probability proportional to the total amount of computational power that it contributes.

2. *Proof-of-stake.* (See Lecture 12 for details.) This approach samples a node running the protocol with probability proportional to the total amount of cryptocurrency that it has locked up in a designated smart contract.

The point is that the quantity that governs a node's probability of selection—the combined computational power of all its identities, or the combined staked cryptocurrencies of all its identities—is independent of how many identities that it has. This is the sense in which proof-of-work and proof-of-stake are sybil-resistance methods.

4 Consensus Protocols vs. Sybil-Resistance Mechanisms

We are not the same. We've talked at length about to types of consensus protocols (BFT-type and longest-chain), and in this lecture and Lecture 12 will talk about two approaches to sybil-resistance (proof-of-work and proof-of-stake). Consensus protocols and sybil-resistance mechanisms are very different concepts, and you should keep them separate in your mind. Conflating them is a common and confusing mistake, and will mark you as a newbie to the technical foundations of blockchains.

For example, think back to the permissioned setting with the PKI assumption that we focused on in Lectures 2–8. Sybil-resistance is trivial in this setting—each node is simply assumed to have a unique public key, and no sybil-resistance method was needed. On the other hand, we had a lot to say about the various consensus protocols you could use in this setting. In the permissionless setting, you now have *two* design decisions to worry about: (i) what underlying consensus protocol to use? (ii) how to prevent Sybil attacks?

[8] Named after the 1973 book *Sybil* about a woman with what was then called a "multiple personality disorder."

Said differently, the purpose of a blockchain's consensus protocol is to decide, given the blocks that have been proposed (and perhaps voted upon) by the protocol's participants, which blocks should be considered finalized (and in what order). As we've seen, there are different approaches to this decision—in a BFT-type protocol a block if finalized if and only if it is accompanied by an appropriate quorum certificate, while in a longest-chain protocol it is finalized if and only if it is sufficiently deep on the longest chain. These two types of protocols are obviously very different, but both exist for the same purpose: to decide which blocks have been finalized.

Sybil-resistance mechanisms address a different question—who has the privilege of proposing and voting on blocks in the first place? Again, this took care of itself in the permissioned setting (with the PKI assumption), where the protocol could hard-code the allowable voters in a BFT-type protocol (namely, all permissioned nodes) and the method of leader selection in a BFT-type or longest-chain protocol (e.g., round-robin or uniformly at random).

Consensus vs. Sybil-Resistance

1. *Sybil-resistance.* Decide which nodes have the privilege of participating, at which times and in what roles.

   [In the permissioned + PKI setting, hard-coded into the protocol.]

2. *Consensus.* Give the block proposals (and possibly votes) of the participating nodes, which ones should be considered finalized?

For example, a common rookie mistake is refer to "proof-of-stake consensus." This phrase doesn't typecheck, because "proof-of-stake" refers to a sybil-resistance mechanism, not to a consensus protocol. It does make sense to speak about a proof-of-stake *blockchain*, meaning a permissionless blockchain protocol that uses proof-of-stake for sybil-resistance (and something else for consensus). Proof-of-stake blockchains come in many flavors—many use BFT-type consensus, a few use longest-chain consensus, and some use still other approaches to consensus.

Pairing proof-of-work with longest-chain. The discussion above hopefully makes it crystal clear that consensus protocols and sybil-resistance mechanisms are two very different things. Accordingly, if we focus on the two dominant methods of consensus (BFT-type and longest-chain) and the two dominant methods of sybil-resistance (proof-of-work and proof-of-stake), we can mix and match to get four different possibilities (Figure 1).[9] Are any of the four possibilities better or more natural than the others?

It's become increasingly clear over the past few years that there is a fairly natural pairing between the two approaches to consensus and the two approaches to sybil-resistance. For proof-of-work, Nakamoto got it exactly right in the original Bitcoin protocol: it pairs

[9]Many but not all of the leading blockchain protocols can be categorized as one of these four types. Some use alternative approaches to consensus (e.g., Avalanche) and others use alternative approaches to sybil-resistance (e.g., Chia).

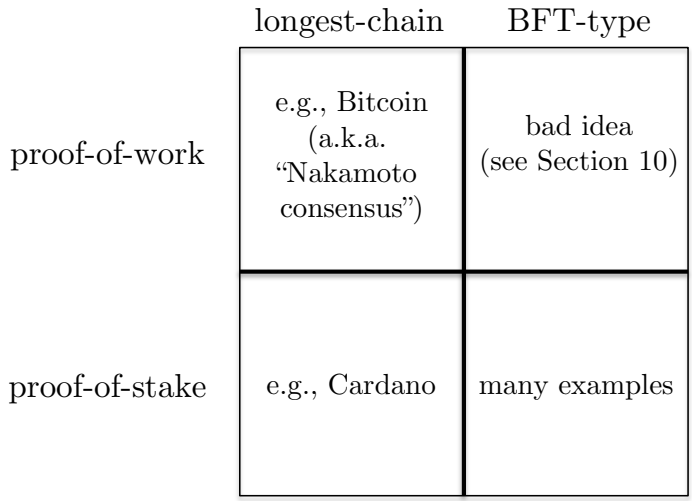

Figure 1: The two most common approaches to consensus and to sybil-resistance combine for four categories that together capture many (but not all) of the leading "layer-1" blockchain protocols.

perfectly with longest-chain consensus, and (as we'll see formally in Section **??**) is largely incompatible with BFT-type consensus protocols.[10] Thus, given that Nakamoto had decided to use proof-of-work to overcome Sybil attacks, they had no choice but to invent a new approach to consensus. You sometimes hear the combination of longest-chain consensus with proof-of-work sybil-resistance referred to as *Nakamoto consensus*, and we'll get into the details of it starting in the next section. As of this writing, the only major blockchain protocols that use Nakamoto consensus are Bitcoin and its forks (like Bitcoin Cash, Litecoin, and Dogecoin).[11]

Pairing proof-of-stake with BFT-type consensus. The story for proof-of-stake sybil-resistance is more nuanced. Most initial experiments with proof-of-stake attempted to use it as a drop-in replacement for the proof-of-work part of the Bitcoin protocol. As we'll discuss in detail in Lecture 12, switching to proof-of-stake sybil-resistance causes a surprising number of complications in longest-chain protocols.[12] It's not impossible to pull off a proof-of-stake longest-chain protocol—there's no impossibility result akin to the one in Section **??** for proof-of-work BFT-type protocols—but it's a lot messier than you'd think. As of this writing, the biggest (by market cap) proof-of-stake longest-chain blockchain protocol is Cardano.

[10] The basic issue is that undetectable fluctuations in the total computational power devoted to the protocol can cause liveness failures, even in the synchronous setting.

[11] Ethereum used Nakamoto consensus for many years, but in mid-2022 switched to proof-of-stake sybil-resistance and a hybrid longest-chain/BFT-type consensus protocol. (Dogecoin may follow suit, we shall see.)

[12] The distinction in Lecture 8 between the stronger assumption (A4') (which holds for proof-of-work blockchains) and the weaker assumption (A4) (the only one enforceable in proof-of-stake blockchains) is one example, but there are a number of others, as well.

The challenges of proof-of-stake longest-chain consensus design, along with the promise of deterministic rather than probabilistic finality, have led to a shift in recent years toward proof-of-stake BFT-type protocols (which at the time of this writing is the most common of the four combinations among major "layer-1" protocols).

When you think about it, BFT-type protocols couple quite naturally with proof-of-stake sybil-resistance. Arguably the most straightforward way to implement proof-of-stake-based protocol participation is to require participants (identified by their public keys) to lock up capital (in the form of the blockchain's native cryptocurrency) in a designated smart contract for a prescribed period of time. Once the participating public keys are known and publicly visible in the staking contract, we are more or less in the permissioned setting (with public keys playing the roles of nodes) with the PKI assumption. One can then run a BFT-type protocol in which the participating "nodes" are the registered public keys; the voting power of a registered public key should be proportional to its stake, so that no one can artificially and costlessly inflate their control over the protocol by creating a large number of public keys. We'll talk much more about the details and nuances of this approach in Lecture 12.

5 Proof-of-Work and Nakamoto Consensus

For us, the main point of proof-of-work sybil-resistance will be to extend the permissioned version of longest-chain consensus and its guarantees in Lecture 8 to an analog ("Nakamoto consensus") in the permissionless setting.

5.1 Lecture 8 Recap

We concluded Lecture 8 by isolating the properties of longest-chain consensus that were really driving the analysis, and it's worth recapping those here. Recall from that lecture that the abstract description of longest-chain consensus includes an underspecified leader selection step (step (2a)), in effect assuming that some mapping from rounds to rounds' leaders is carried out by some black box (Figure 2).[13]

Key properties for the analysis. Three properties of this black box were necessary for the proofs of the basic consistency and liveness properties in Lecture 8, none of which are fundamental to the permissioned setting per se:

Required Properties of the Leader Selection Box

1. (Same as assumption (A2) from Lecture 8) It is easy for all nodes to

[13]Step (1) is to start with a genesis block, which is known to all nodes at and only at the time of the protocol's commencement (a trusted setup assumption, called (A1) in Lecture 8). The protocol proceeds in rounds, and in each round there is a single leader node (selected in step (2a)). In Step (2b), the leader of the round proposes blocks and explicit predecessors for those blocks. Honest nodes are instructed to propose a single block that extends the longest chain that they are aware of, breaking ties arbitrarily.

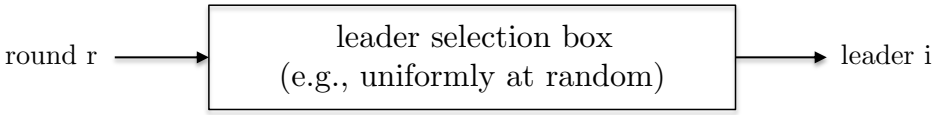

Figure 2: Step (2a) of longest-chain consensus is effectively a black box that maps round numbers to leader nodes.

verify whether a given node is the leader of a given round.

2. (Same as assumption (A3) from Lecture 8) No node can influence the probability with which it is selected as the leader of a round.

3. (Required hypothesis for the common prefix property, liveness, and chain quality guarantees in Lecture 8) The sequence of leaders is sufficiently balanced.[14]

"Sufficiently balanced" means different things, depending on the context (see Lecture 8 for details), but there's a single sufficient condition that guarantees (with high probability) all of the variants that we care about:

Key Sufficient Condition for Consistency and Liveness of Longest-Chain Analysis

Leaders in different rounds are chosen independently and, in each round, the probability that the leader is Byzantine is at most $\alpha < \frac{1}{2}$.

In other words, if you push the button on the black box to generate a new leader, the leader that pops out is more likely to be honest than Byzantine (with each selection independent of previous ones). This condition drove all the "proportional representation" arguments that were used throughout Lecture 8, and given that the condition holds the consequent arguments do not depend at all on being in the permissioned setting.

The missing ingredient. All the properties of longest-chain consensus that we care about thus boil down to having a sufficiently balanced leader sequence, and this in turn boils down to making sure that the key sufficient condition above holds. In the permissioned setting with the PKI assumption, our solution was to assume that less than half the nodes are Byzantine and to choose leaders independently and uniformly at random. For a permissionless solution, we'll need a different method of (sybil-resistant) random sampling (i.e., proof-of-work) and a different assumption (that less than half of the total computational power is contributed by Byzantine nodes). But as long as we manage to somehow satisfy the sufficient condition above in the permissionless setting, we'll be good to go, with all the consistency and liveness

[14] In longest-chain consensus, all honest nodes are interchangeable (they all behave identically) and all Byzantine nodes are interchangeable (they are colluding anyway). Thus all that matters about a leader sequence is its pattern of H's and A's.

guarantees from Lecture 8 carrying over (assuming, as usual, that the security parameter k controlling the depth-til-finalization is chosen appropriately large).

In summary, the protocol and analysis of Lecture 8 is missing one and only one ingredient from a permissionless protocol with the same consistency and liveness guarantees: a permissionless leader selection box that, under an appropriate assumption about the resources of the Byzantine nodes, selects honest leaders more frequently than Byzantine ones. Section 5.2 next will supply exactly this missing ingredient.

Nakamoto consensus vs. abstract longest-chain consensus. As an advance warning, three aspects of the proof-of-work version of longest-chain consensus may surprise you, given the abstract version that we've been focused on thus far. First, steps (2a) and (2b)—the choice of a leader and that leader's choices of blocks and block predecessors—will be smooshed together into a single step, in effect with each node making its step (2b) choices before knowing whether it's selected as the round's leader. Second, as we'll see in Section 5.3, the way the two steps will be smooshed together will actually prevent any leader from ever proposing more than one block in a single round (i.e., as promised, assumption (A4') from Lecture 8 will be satisfied). Finally, you might be thinking of each round as having some fixed duration like 10 seconds. In the proof-of-work version of longest-chain consensus, new rounds are triggered by random events and hence a round's duration is a random variable. By contrast, none of these three idiosyncrasies are present in proof-of-stake versions of longest-chain consensus (see Lecture 12).

5.2 What Is Proof-of-Work?

The basic idea. So how does proof-of-work, uh, work? The basic idea is to declare the leader of the next round to be the first node that manages to come up with a solution to a hard puzzle. It should seem plausible that, for an appropriate choice of puzzle, your likelihood of being the first solver of the puzzle should depend on the total amount of computational effort that you put in to solving it (with the number of public keys that you control totally irrelevant).

Even at this vague level of description, you should be able to see what I meant in Lecture 8 when I said that, in the proof-of-work version of longest-chain consensus, rounds are "event-driven" (with the event being that some node successfully solves a hard puzzle). Similarly, you see why (as mentioned above) rounds will have variable and random durations (depending on whether some node gets lucky and finds a puzzle solution quickly, or if all nodes are unlucky and take a long time to find a puzzle solution).

In accordance with the permissionless setting and an unknown set of nodes, this approach to randomly sampling a leader (with the leader whichever node happens to get lucky first) is "bottom-up" (implemented by the nodes themselves through a process external to the protocol) rather than "top-down" (implemented in-protocol, as in the permissioned version of longest-chain consensus).

A little history. Proof-of-work actually dates back to 1992, seventeen years before the Bitcoin protocol dropped. Dwork and Naor [**?**] introduced the idea in the context of spam-fighting, with the goal of making the sending of an email modestly computationally expensive (e.g., taking 0.1 second or 1 second per email on a commodity machine). This was a pretty prescient paper—not only because blockchains didn't exist yet, but (believe it or not) spam email didn't really exist yet either! Nakamoto, who was aware of [**?**] and also Back's later use of the idea in HashCash [**?**], was the first to apply the proof-of-work technique in the design of consensus protocols. As of 2022, permissionless consensus is clearly the killer application of proof-of-work.

Cryptographic hash functions. What do I mean by a "hard puzzle"—like, the Saturday New York Times crossword? One could imagine various approaches, but there's one approach that works really, really, well—the same approach taken by Nakamoto in the original Bitcoin paper [**?**], where the puzzle is to approximately invert what's called a "cryptographic hash function."

To this point in the lecture series, we've adopted only one cryptographic assumption—the relatively uncontroversial assumption that secure digital signatures schemes exist (see Lecture 1 for extended discussion). That is, it's possible to sign a message (using a private key) such that anyone can verify the signature (using the corresponding public key), and such that it's impossible (for all practical purposes) to forge such signatures without knowledge of the private key. As we've seen (e.g., in Tendermint), such signature schemes are very useful in the design of (permissioned) consensus protocols (especially when all nodes' public keys are common knowledge as the start of the protocol, a.k.a. the PKI assumption).

Now we're make a second cryptographic assumption—another one that is relatively uncontroversial in practice—namely that *cryptographic hash functions (CHFs)* exist.[15] A hash function, remember, is just a function from some domain to some range. Depending on the context, hash functions can be use to compress or expand the input. In the context of key-value stores implements as hash tables, the point of a hash function is to spread data evenly throughout the hash table's array (cf., "universal hashing").

Cryptographic hash functions are designed with a different goal in mind, which is to be totally inscrutable, meaning (for all practical purposes) totally unpredictable. Ideally, whatever you feed in as input, you get back as output some intelligible gibberish that you've never seen before in your life.

If you want to remember just one function that is believed to be cryptographic in this sense, remember "SHA-256." (Here "SHA" stands for "secure hash algorithm" and the "256" refers to the number of bits of output. The input to SHA-256 can be of any length.) I encourage you to look up a reference implementation of SHA-256 on the Web—it's not that long, maybe 100 lines of C code. Despite being a deterministic function whose source code is known to everybody, it's a totally inscrutable function. For all practical purposes,

[15] In this lecture, we'll use CHFs only to implement proof-of-work sybil-resistance. Later in the lecture series, they will be indispensable tools for a totally different reason, in the construction of cryptographic commitment schemes (Merkle trees, etc.).

the output of SHA-256 is completely unpredictable for any input that you haven't already evaluated it on.

The random oracle assumption. We'll formalize the idea of an inscrutable and unpredictable cryptographic hash function via the *random oracle assumption*. This assumption will on the one hand be patently false, but in other ways a surprisingly close approximation of reality.[16] The assumption states that the hash function we're using (such as SHA-256) may as well be a random function, in the sense that no program that anyone writes will ever be able to tell the difference. Here by "random function" we mean literally a function chosen uniformly at random from the set of all functions with the given domain and range; or, equivalently, a function for which the output of each input is chosen independently and uniformly at random from the range (like $\{0,1\}^{256}$).

I encourage you to think about a random function as being defined by lazy evaluation, on a need-to-know basis. Think of it as a box with a little gnome in it. When you evaluate the hash function h for the first time on some input x, the gnome in the box flips a coin 256 (say) times and records and outputs the resulting string of 256 zeroes (heads) and ones (tails).

If you ever re-evaluate the hash function at the same input x, the gnome in the box checks its records and produces the same output $h(x)$ as before. But if you feed in some different input y, the gnome in the box will flip a fresh set of 256 coins, recording and outputting the resulting 256-bit string $h(y)$. It should be clear that such a function is completely unpredictable—the only way to learn anything about its output on a given input is to actually evaluate it on that input, and learning the outputs corresponding to a bunch of inputs tells you nothing about the function's outputs for other inputs.

The assumption that some concrete deterministic function is the same as a random function is, of course, patently false. SHA-256, for example, is 100 lines of code, not a gnome in a box. But here's the thing: deterministic phenomena can appear random when there is a bounded amount of computational power. For example, imagine you're watching the coin toss at the start of a sporting event, and you try to guess the outcome of the toss while the coin is spinning rapidly while six feet up in the air. It's hard to imagine how you would predict the actual outcome with better-than-50% probability. On the other hand, if you had a much more computationally powerful system than just your naked eye—tons of crazy cameras tracking the coin from different angles, a super-computer dedicated to fine-grained physical simulations, etc.—it's conceivable that you could predict the outcome of the coin flip close to 100% of the time. Similarly, while SHA-256 is in principle completely predictable to a sufficiently powerful (but likely unrealizable) computer, it may well appear random to any computation that we can realistically carry out.

There is also some chance that someone will come up with a new and practical algorithm for better-than-random prediction of a concrete function like SHA-256, and such an event would likely have detrimental consequences for proof-of-work blockchains like Bitcoin. Of course, there's also some chance that someone will come up with an efficient algorithm for

[16] For a good and lengthy discussion of this seeming paradox, see [**?**].

the factoring or discrete logarithm problems, in which case all hell will break loose (at least in the short term). Just as we've been happy to assume that there's no fast factoring or discrete logarithm algorithm right around the corner, so too will we assume that better-than-random prediction of SHA-256 won't happen anytime soon. This assumption grows more battled-tested as the years roll by, and this is often all you can hope for when it comes to cryptographic assumptions.

From CHFs to proof-of-work puzzles. Now let's take on faith that we have a concrete function h like SHA-256 which is practically indistinguishable from a random function. How can we use such a function to define hard puzzles? The canonical way is, given a "difficulty threshold" τ:

Canonical Proof-of-Work Puzzle

find an input x such that $h(x) \leq \tau$.

For concreteness, assume that the hash function h produces 256-bit outputs, which we then interpret as nonnegative integers (written in binary).

The parameter τ is a tunable difficulty parameter.[17] The hardest version of this puzzle is when $\tau = 0$, in which case the task is to invert the function h at the all-zeroes output. The easiest version is when $\tau = 2^{256}$, in which case every input qualifies as a solution. If $\tau = 2^{176}$, for example, the puzzle is to find an input x such that $h(x)$ has at least $(256 - 176) = 80$ leading zeroes. In general, bigger τ's mean easier puzzles (i.e., with more x's qualifying as solutions), and smaller τ's mean harder puzzles.

The fact that these types of puzzles have such precisely tunable difficulty is a big part of why they form the basis for all proof-of-work blockchain protocols. For an NP-hard problem like satisfiability or the traveling salesman problem, for example, it's not at all clear what it would mean to "double the instance difficulty." (Whereas with the puzzles above, this would mean cutting the value of τ in half.)

The optimality of repeated guessing under the random oracle assumption. Let's tie this section's threads together by investigating the "partial CHF inversion" puzzles above under the random oracle assumption. Suppose, for example, that we're working with SHA-256 and a difficulty threshold of 2^{176}. The puzzle is then to find an input x such that the output of SHA-256 starts with 80 zeroes in a row.

Now let's adopt the random oracle assumption, and model SHA-256 as a random function h—a gnome in a box. The game is now to find an input x such that $h(x)$ starts with 80 zeroes—i.e., such that the first 80 coin flips by the gnome come up heads. Each time h is queried on a new input, there's a one in 2^{80} chance that the first 80 coin flips are heads. That is, the probability that any given input happens to be a puzzle solution is 2^{80}. Moreover, under the random oracle assumption, the only way you can learn anything about h

[17] As we'll see in Section 7, in practice τ is generally tuned to target a specific rate of block production (e.g., in Bitcoin, a target of one block per 10 minutes).

is through querying it at various inputs, and its outputs-so-far tell you absolutely nothing about its outputs on as-yet-unevaluated inputs.

What this means is that, in searching for a puzzle solution, there are literally no options available other than repeatedly evaluating the hash function on some sequence of inputs $x_1, x_2, x_3, \ldots$ until you get lucky and stumble on an input x_i with $h(x_i) \leq \tau$. (You can think of repeatedly throwing darts at a dartboard that has a really, really small bullseye.) And because each input is equally likely to be a puzzle solution (e.g., probability 2^{-80}), your probability of success is the same no matter what sequence of inputs you choose to evaluate (as long as they are distinct).

With the parameter choices in our running example, it's really, really hard to find a puzzle solution. With each guess having only a 2^{-80} of succeeding, you would expect to find a puzzle solution only after trying 2^{80} different guesses (of course, it may be much more or much less, depending on your luck). This would take forever on conventional computers, and the only hope of ever finding a puzzle solution would be to throw an enormous amount of specialized hardware at the problem (so as to explore the input space in a massively parallel way). If τ were 2^{226}, and so a success probability of 2^{-30} per guess, a commodity laptop could find a puzzle solution without too much trouble.

To summarize: (i) if we assume that SHA-256 is indistinguishable from a random function, the only way to find a puzzle solution is through repeated guesses of distinct and otherwise arbitrary inputs; (ii) if each guess is a puzzle solution with probability p, then on average $1/p$ guesses are required to identify a solution.[18]

Whatever your doubts about the random oracle assumption, the upshot of the argument above—that in a proof-of-work blockchain, nodes running the protocol will find puzzle solutions via naive repeated guessing—is a remarkably accurate of current practice.

5.3 What Should Puzzle Solutions Encode?

We now understand that, in the proof-of-work version of longest-chain consensus, the leader of a round (step (2a) in the abstract description) is defined as the first node to find a puzzle solution, meaning identify an input x such that $h(x) \leq \tau$, where h is a cryptographic hash function (like SHA-256) and τ is a difficulty parameter (like 2^{176}) that is somehow determined by the protocol.

Committing to block proposals and predecessors. Next, let's drill down on the requirements for a puzzle solution x. Thus far, we've only required that it hashes to a sufficiently small number, but nothing's stopping us from imposing some additional conditions, for example requiring x to conform to a prescribed format. Specifically, it will be convenient to require x to encode a node's choices of blocks and predecessors in step (2b) should the node happen to be elected as the next leaders—that is, should x happen to hash to a number that's τ or less. Thus, a node will learn whether it's the leader of the next round only after

[18] Technically, this is that fact that the expected value of a geometric random variable with parameter p (i.e., the number of coin flips requires to get a "heads," when the probability of heads is p) is $1/p$; see e.g. `https://en.wikipedia.org/wiki/Geometric_distribution`.

it has committed to its step (2b) choices and packaged them into an input x—under the random oracle assumption, the node can't predict whether $h(x) \leq \tau$ until it has actually evaluated h on x. (This is the "smooshing together" of steps (2a) and (2b) of longest-chain consensus that was advertised in Section 5.1.) In the other versions of longest-chain consensus (the permissioned version in Lecture 8 and the proof-of-stake version in Lecture 12), by contrast, a leader is generally free to choose blocks and predecessors after it has been selected as the leader of a round.

Forcing a single block proposal and predecessor. But why stop there? In the abstract description of longest-chain consensus, a round's leader is free to propose multiple blocks. (For example, a Byzantine node could propose one block to some of the honest nodes and a conflicting block to the remaining honest nodes.) Honest leaders, meanwhile, are instructed to propose only a single block (extending the longest chain). Given that we're already forcing a node to commit to its step (2b) choices in any candidate puzzle solution, why not automatically disqualify any proposed input x that names more than one block proposal? In other words, let's deem an input x a puzzle solution if and only: (i) x encodes a single block proposal and predecessor; and (ii) $h(x) \leq \tau$. This makes it impossible for a leader (Byzantine or otherwise) to propose multiple blocks in a single round. (This is the third comment from Section 5.1, promising that the stronger assumption (A4') from Lecture 8 would hold for Nakamoto consensus.) Further proposals can be made only in future rounds in which the same node is again the round's leader. That requires solving an entirely new puzzle; under the random oracle assumption, solving one puzzle provides no advantage to solving a different one.

Format for puzzle solutions. Conceptually, we can think of a candidate puzzle solution x as having multiple fields. Two of the fields should contain a block proposal B and a predecessor *pred*.[19] You can also think of there being a third field that contains the node's public key pk (or at least, one of its public keys) so that the node can claim credit for the puzzle solution.[20]

Summarizing, the tentative proposal to require inputs x of the form $B||pred||pk$, where "||" denotes concatenation. Remember, in addition to satisfying these formatting requirements, an input x qualifies as a puzzle solution only if its hashes to something small (i.e., $h(x) \leq \tau$). We would therefore expect a motivated node running the protocol to experiment with different x's—i.e., with different blocks, and/or different predecessors, and/or different public keys that it can costlessly generate—in its quest for a (properly formatted) input that with a sufficiently small hash. Such experimentation is sometimes called *grinding* (as a node is grinding through many candidate solutions, looking for a valid one).

[19] In this lecture we won't worry about the details of how these are represented. When we study the Bitcoin and Ethereum protocols in more detail later in the lecture series, we'll see that B is generally a cryptographic commitment to a set of transactions (a "Merkle root") rather than the transactions themselves, and that *pred* typically refers to an existing block via a hash of that block's header (i.e., metadata).

[20] The motivation for this will become obvious next lecture when we introduce cryptocurrency rewards for block production. In practice, the public key of a round leader is included in its block only indirectly, via what's called a "coinbase transaction."

Given nodes' inevitable experimentation with different x's, it's convenient to include a fourth field, called a "nonce" (which stands for "number used once"). The nonce is just a bunch of free bits with no purpose other than providing nodes with degrees of freedom for generating lots of different well-formatted inputs (without needing to, for example, grind over public-private key pairs).

Format for Puzzle Solutions

$$B||pred||pk||nonce$$

Your mental model for a node running a proof-of-work blockchain protocol should be that it fixes a block B of transactions, a predecessor *pred* for that block, its public key pk, and then grinds through nonces *nonce* until $h(B||pred||pk||nonce) \leq \tau$. (If it hears about a new block in the meantime, produced by some other node, it will likely update its choices of B and *pred* and restart the nonce grinding process.)[21] This is a surprisingly accurate description of how proof-of-work blockchains like Bitcoin work in practice.[22]

6 Properties of Proof-of-Work & Nakamoto Consensus

Now that we understand how proof-of-work works, we can return to fulfilling our earlier promises. This section proves that proof-of-work is indeed resistant to sybil attacks (under the random oracle assumption), that it provides the missing "permissionless black box" needed to extend longest-chain consensus from the permissioned to the permissionless model, and notes some additional remarkable properties possessed by Nakamoto consensus.

6.1 Sybil-Resistance

The most important property of proof-of-work for our purposes is its *sybil-resistance*, meaning that the probability that a node is selected as the next leader—i.e., is the first to solve a hard puzzle, or specifically to partially invert a cryptographic hash function—is independent of the number of public keys that it might control.

For the analysis, we will adopt the random oracle assumption of Section 5.2—that whatever hash function we're using (e.g., SHA-256) is indistinguishable from a random function, in which a gnome in a box flips a fresh set of 256 coins every time a new input is evaluated. We will also assume that every node running the protocol repeatedly tries different

[21] Technically, in proof-of-work blockchains like Bitcoin one often differentiates between the nodes that are grinding away and striving to be selected as a leader and to propose a new block (usually called *miners*, as if digging for gold (i.e., block rewards)) and the nodes that are merely maintaining the blockchain and checking that miners are following the protocol's rules (usually called *full nodes* or *validators*). For example, if you're building on top of a blockchain and want to query the blockchain's state from a smart contract, you care about talking to a validator, not a miner. For simplicity, we're currently assuming that every node running the protocol is acting as both a miner and a validator.

[22] We'll discuss some additional Bitcoin-specific details of the puzzle solution format and the block production process later in the lecture series, during our Bitcoin deep dive.

nonces until some gets lucky and finds a nonce that (coupled with the node's choice of block, predecessor, and public key) hashes to a number that is at most the current difficulty threshold τ.[23] As discussed in Section 5.2, under the random oracle assumption, this the obvious and optimal strategy for the nodes running the protocol.

Under these assumptions, the probability that a given node is selected as the next leader depends only on the total amount of computational power that it contributes to the protocol, independent of the number of identities it might control. Formally:

Theorem 6.1 (Sybil-Resistance of Proof-of-Work) *Suppose n nodes $1, 2, \ldots, n$ make $\mu_1, \mu_2, \ldots, \mu_n$ distinct nonce guesses per second. Then, for every node i and round r,*

$$\mathbf{Pr}[\textit{node } i \textit{ is the round-}r \textit{ leader}] = \underbrace{\frac{\mu_i}{\sum_{j=1}^{n} \mu_j}}_{\substack{i\textit{'s fraction of} \\ \textit{the overall hashrate}}}. \tag{1}$$

Moreover, selections of leaders in different rounds are independent.

The proof of Theorem 6.1 is just some easy probability that follows from the random oracle assumption, as we'll see below. Before that, a few comments. First, the μ_i's are often referred to as the nodes *hashrates*, reflecting the fact what we care about a node's "computational power" only inasmuch as we care about the rate at which it can try out different nonces (which one hash function evaluation necessary and sufficient to check if a given choice yields a valid puzzle solution).[24] Second, while the statement of Theorem 6.1 references a set of n nodes for notational purposes, we stress that the proof-of-work process works in the permissionless setting and so is independent of what this node set might be. At any given time, there will be some set of nodes running the proof-of-work protocol and searching for puzzle solutions, those nodes will have some hashrates at that time, and Theorem 6.1 will then apply to those particular nodes and hashrates at that time.[25] Finally, because the right-hand side of (1) is independent of how many identities node i controls, Theorem 6.1 implies that proof-of-work is indeed sybil-resistant.

Corollary 6.2 (Sybil-Resistance of Proof-of-Work) *The probability that a node is selected as a leader depends only on its share of the overall hashrate, independent of its number of identities.*

[23] We will generally assume that all the guesses made by all the nodes are distinct. This would be true if, for example, each node uses a different public key and uses distinct nonces in its guesses.

[24] For example, the specialized hardware (ASICs, for "application-specific integrated circuits") used for Bitcoin mining are not general-purpose computers and are optimized to evaluate SHA-256 as quickly as possible (at the expense of all other functionality).

[25] By "node," we mean an actual physical machine running the protocol, not a specific public key. For example, if a node is capable of evaluating one billion nonces per second, this will be true whether all the guesses are concentrated under a single pubic key or split 50/50 between two different public keys. Spinning up multiple threads under different identities doesn't magically increase the number of guesses that the machine can make.

Looking at Theorem 6.1 and Corollary 6.2, it should be clear that a node i *can* increase its probability of selection by beefing up its machine and increasing its hashrate μ_i (assuming other nodes' hashrates stay fixed, this would increase node i's share of the overall hashrate). But at least this is a *costly* manipulation (more computers $\Rightarrow$ more money), unlike sybil attacks, which are essentially costless (repeatedly enter `ssh-keygen` at a Unix prompt). You might have mixed feelings about divvying up control over a blockchain protocol according to the nodes' financial investments in hashrate, but this (or something similar with a different costly resource) appears to be necessary to achieve sybil-proofness in a permissionless setting.

The analysis. The proof-of-work sybil-proofness property (Theorem 6.1) may seem intuitive—if one machine is making guesses twice at rapidly as another one, presumably it's twice as likely to be the first node to find a puzzle solution. If you find this argument convincing, feel free to skip to Section 6.2. But given the importance of this property, it seems appropriate to provide a more formal proof of it (under the random oracle assumption).

Proof of Theorem 6.1: Fix an arbitrary node $i \in \{1, 2, \ldots, n\}$. Consider two events (i.e., things that might or might not happen): event A, that a given guess was made by node i; and event B, that a given guess turns out to be a valid puzzle solution (and therefore kicks off a new round). The theorem statement concerns the conditional probability $\mathbf{Pr}[A|B]$—the probability that the guesser of a puzzle solution (i.e., the leader of round) is the node i.

Under the random assumption assumption, every guess made by a node produces a fresh string of perfectly random bits (the coin flips by the gnome in the box). This means that the probability that any given guess is a valid puzzle solution is the same no matter which node proposed it. (As all nodes face the same difficulty parameter τ.) In particular, event B is independent of event A: $\mathbf{Pr}[B|A] = \mathbf{Pr}[B]$. Independence of events is symmetric, to event A must then be independent of event B: $\mathbf{Pr}[A|B] = \mathbf{Pr}[A]$. (Formally, this is a special case of Bayes' rule.) The probability $\mathbf{Pr}[A]$ that the current guess was made by node i equals the relative frequency of guesses made by node i, which is $\mu_i / \sum_{j=1}^{n} \mu_j$, as promised.

Finally, under the random oracle assumption, puzzle-solving efforts from past rounds have no bearing on future rounds (with all nodes continuing to repeatedly try out different guesses, each equally likely to succeed). Thus, leaders of different rounds are selected independently. ■

6.2 Permissionless Consensus with Provable Guarantees

Supplying the missing ingredient. Now let's segue from studying proof-of-work in its own right and pick up the story where we left off at the end of Section 5.1, examining the composition of proof-of-work sybil-resistance with longest-chain consensus (i.e., Nakamoto consensus). Back in that section, we agreed that key to the analysis in Lecture 8 (which established several probabilistic consistency and liveness properties for longest-consensus) was the property that leaders in different rounds are chosen independently and, in each round, the probability that the leader is Byzantine is at most $\alpha < \frac{1}{2}$. This property drove all the "proportional representation arguments" that implied (with high probability) various

balancedness properties of the generated leader sequence, and these balancedness properties in turn implied the consistency and liveness guarantees (provided the security parameter k is chosen sufficiently large, given the bound α on the probability of a Byzantine leader and the duration of interest). We barely used the power of the permissioned model in these arguments—only for implementing the leader selection black box, e.g. using a round-robin order or uniformly random leaders.

In Nakamoto consensus, rounds' leaders are chosen using proof-of-work (with the leader of the next round the first node to partially invert a cryptographic hash function, and with their proposed block and predecessor encoded in the solution). Proof-of-work supplies the missing permissionless ingredient and turns longest-chain consensus is to a purely permissionless consensus protocol—the protocol operates without any knowledge of the set of nodes (with the nodes themselves rather than the protocol selecting leaders, in effect) and with all communication via broadcast. The sybil-resistance property of proof-of-work (Theorem 6.1) implies that the probability that a given node is the leader of a given round equals its share of the overall hashrate, with leaders of different rounds chosen independently. By extension, the probability that the leader of a given round is a Byzantine node equals the fraction of the overall hashrate that is controlled by such nodes. Thus, as long as this fraction α is less then $\frac{1}{2}$, we should be good to go! (This $< 50\%$ Byzantine hashrate assumption replaces our old assumption from the permissioned model of $< 50\%$ nodes.)

Checking the assumptions. Some chores remain before we can declare victory with a permissionless consensus protocol with provable guarantees. Specifically, the analysis of longest-chain assumption relied on five assumptions (above and beyond the permissioned model and the PKI assumption), and we need to check that our proof-of-work implementation of longest-chain consensus doesn't violate any of them so that that lecture's analysis continues to apply to Nakamoto consensus. For reference, here's a restatement of those assumptions from Lecture 8:

Assumptions (A1)–(A5)

(A1) No node has knowledge of the genesis block prior to the deployment of the protocol.

(A2) It is easy for all nodes to verify whether a given node is the leader of a given round.

(A3) No node can influence the probability with which it is selected as the leader of a round in step (2a).

(A4) Every block produced by the round-r leader must claim as its predecessor some block that belongs to a previous round.

(A5) At all times, all honest nodes know about the exact same set of blocks (and predecessors).

Assumption (A1). Let's take them one at a time. Assumption (A1) was important in our proof in Lecture 8 of the common prefix property. It is a trusted setup assumption—basically, that Byzantine nodes couldn't get started on puzzle-solving until the protocol was deployed—meaning that it's the responsibility of whomever deploys the protocol rather than that of the protocol itself. Thus, in our analysis, we simply assume that its true without worrying about why.

That said, in practice, when deploying a protocol, one can take pains to build up confidence among participants that the trusted assumptions are indeed true. Nakamoto was well aware of Bitcoin's trusted setup assumption, and released the protocol (on January 3rd, 2009) with a genesis block that included a hard-coded message that referenced a very recent headline of the Financial Times (about a bailout for British banks—remember this was in the middle of the Great Recession). Presumably, then, no one (not even Nakamoto) knew what the Bitcoin protocol's genesis block was until at most a few hours before its deployment.

Assumption (A2). The next three assumptions are asserted properties of the protocol (as opposed to an external-to-protocol trusted setup assumption), we need to check that all of them hold. Happily, with proof-of-work sybil-resistance and with puzzle solutions encoding block proposals and predecessors, these all take care of themselves. Let's start with assumption (A2). A round's leader is defined through having a valid puzzle solution. If a node can exhibit such a solution (an input x that is formatted correctly and with $h(x) \leq \tau$), all the other nodes can check its formatting and check for itself that it hashes to something small. (The hash function h is part of the protocol's description and designed to be easy to evaluate, and as we'll see in Section 7, any node following along with the protocol knows what the current difficulty parameter τ is.) Thus if you possess such a puzzle solution, all other nodes can easily recognize you as the next leader. If you don't, there's no way to trick the other nodes into thinking that you are the next leader (whatever you try to supply to them will either be improperly formatted or hash to something bigger than τ, so you'll be caught red-handed by the other nodes).

Assumption (A3). Assumption (A3) follows immediately from the sybil-resistance property of proof-of-work (Theorem 6.1, which relies on the random oracle assumption). The probability that a node is selected as a round's leader equals its fraction of the overall hashrate, and there's nothing that the node can do about it.[26]

Assumption (A4). For assumption (A4), as advertised, Nakamoto consensus actually satisfies the stronger condition (A4') that at most one block is produced in each round. This follows from the fact that there is exactly one puzzle solution per round (by the definition of a round) and that a valid puzzle solution can (by definition) name only a single block proposal (see Section 5.3). Also, such a puzzle solution can only name as a predecessor a block that existed at the time of its creation, which must necessarily have been produced in some previous round.

[26] Other than investing in additional hashing power, which occurs at a slower timescale than the one we're analyzing here; see also the discussion following Corollary 6.2. Thus assumption (A3) might more accurately state that no node can *costlessly* influence the probability with which it is selected as the leader of a round.

Assumption (A5). The final assumption (A5) is an assumption about the communication network (that communication is instantaneous), not the protocol, and so we're not in a position to enforce it. The good news is that this assumption isolated the most difficult aspect of consistency in longest-chain consensus—finality, a.k.a. the consistency of a node with its future self (as opposed to consistency between nodes at a given moment in time, which is trivialized by (A5)). The bad news is that this assumption is clearly false—even in a well-functioning network, messages have nonzero delays. We'd therefore like to relax this assumption to something more plausible—at the very least, to the synchronous model with some nonzero maximum message delay Δ—without materially changing the consistency and liveness guarantees that we've come to expect for longest-chain consensus. And this is exactly what we'll do in Section **??** (under the assumption that Δ is small relative to the expected duration of a round)!

Nakamoto consensus: the final scorecard. Summarizing, here are the key assumptions under which Nakamoto consensus has provable (probabilistic) consistency and liveness guarantees:

Key Assumptions for Consistency and Liveness of Nakamoto Consensus

1. The random oracle assumption—e.g., SHA-256 is for all practical purposes indistinguishable from a random function (see Section 5.2).
2. The trusted setup assumption (A1)—no advance knowledge of the genesis block.
3. Less than half of the hashrate is controlled by Byzantine nodes (by Theorem 6.1, this guarantees that the key sufficient condition from Lecture 8 is satisfied).
4. The communication network conforms to the synchronous model, with finite maximum message delay Δ.
5. The difficulty parameter τ is tuned so that the average duration of a round (i.e., the average amount of time between two successive valid puzzle solutions) is much larger than Δ.[27]

The next two sections (Sections 6.3 and 6.4) discuss some additional remarkable properties of Nakamoto consensus; the time-constrained reader can skip to Section 7 without any loss of continuity.

[27] How much larger? Think 1–2 orders of magnitude. The precise bound on the maximum-allowable fraction of Byzantine hashrate depends on the ratio between the average round duration and Δ, and approaches 50% as this ratio approaches infinity (see Section **??** for details).

6.3 How Did We Elude the PSL-FLM Impossibility Result?

Strong permissionlessness. While Nakamoto consensus does require a trusted setup assumption—assumption (A1), that no node has advance knowledge of protocol's genesis block—it does *not* require our usual trusted setup assumption, the PKI assumption. That is, nodes running the protocol require no knowledge of the public keys possessed by the other nodes running the protocol. For example, to join the set of nodes running the Bitcoin protocol, all you need to do is download some software, sync up your machine to the current state of the blockchain, and start following along (maintaining the blockchain, trying find puzzle solutions, etc.). No one else knows that you exist!

While there's no single widely accepted definition of a "permissionless" protocol, no matter what definition you use, Nakamoto consensus certainly qualifies. At any moment in time, a new block might well get produced under a public key that nobody has ever seen before. (Once that block is published, everybody sees the public key responsible for it. But whomever published the block can generate and switch to using a new public key immediately, thereby returning to puzzle-solving in complete obscurity.) As we'll see in Lecture 12, blockchains based on proof-of-stake sybil-resistance tend to be "somewhat less permissionless," in that the set of public keys eligible for producing the next block is usually maintained in public view (i.e., the set can be derived from the blockchain's current state).

Review: the PSL-FLM impossibility Result. If you've been following this lecture series really, really closely, you might at this point have a question, namely why doesn't Nakamoto consensus contradict the impossibility results for SMR protocols that we saw earlier in the series.

Back in Lecture 2, we studied the Dolev-Strong protocol, which solves the Byzantine broadcast problem (satisfying termination, validity, and agreement) even when 99% of the nodes are Byzantine, under two key assumptions: (i) the synchronous model; (ii) the PKI assumption. (Iterating this protocol then gave us a SMR protocol satisfying consistency and liveness.) Later lectures probed to what extent assumptions (i) and (ii) are necessary for this result. For example, in Lecture 6 we saw that, in the partially synchronous model, there's no solution to the Byzantine broadcast problem when at least one third of the nodes are Byzantine (even with the PKI assumption).

Relevant here, though, is our probing of assumption (ii) in Lecture 3. The main result there—you might remember the thought experiment on the hexagon, and the various ways that a Byzantine node might simulate simultaneously four fictional honest nodes—states that without the PKI assumption, even in the synchronous model and even assuming that cryptography exists, there is no protocol that solves the Byzantine broadcast problem when at least one-third of the nodes are Byzantine. Thus, the advance distribution of nodes' public keys was an unavoidable requirement of the Dolev-Strong protocol.

Nakamoto consensus, meanwhile, can solve the state machine replication (SMR) problem even when more than one third (up to 49%) of the hashrate is Byzantine. Why doesn't this guarantee (without PKI, consensus is possible with 49% Byzantine) contradict the PSL-FLM impossibility result (without PKI, consensus is impossible with 34% Byzantine)?

Some false resolutions. The answer is subtle, so let's step through some incorrect answers before arriving at the correct resolution.

> *Guess #1: the PSL-FLM impossibility result only limits the fraction of* nodes *that are Byzantine, while the Nakamoto consensus guarantees require that less than half the* hashrate *is controlled by Byzantine nodes.*

The "node vs. hashrate" distinction is irrelevant. Our guarantees for Nakamoto consensus hold for any distribution of hashrates (as long as less than half of the total is controlled by Byzantine nodes), so they hold in particular when every node happens to have exactly the same hashrate. In this special case, the "$< 50\%$ hashrate" assumption becomes a "$< 50\%$ nodes" assumption, which would seem to contradict the one-third threshold established by the PSL-FLM impossibility result.

> *Guess #2: the PSL-FLM impossibility result is for the Byzantine broadcast problem, while the Nakamoto consensus protocol solves the SMR problem.*

This is a better guess, but still incorrect. The hexagon argument from Lecture 3 can in fact be extended to hold also for the Byzantine agreement and SMR problems (see the Appendix of that lecture). Alternatively, Nakamoto consensus can be extended to solve the Byzantine agreement problem when 49% of nodes/hashrate are Byzantine (even without the PKI assumption) [**?**].

> *Guess #3: the PSL-FLM impossibility result is for deterministic protocols that guarantee safety and liveness, while the Nakamoto consensus protocol is randomized and guarantees only probabilistic safety and liveness.*

Another good guess, but again not the correct resolution. As it turns out—an excellent, tricky homework problem—the proof of the PSL-FLM impossibility result in Lecture 3 can be extended to rule out randomized protocols for Byzantine broadcast (or agreement) that achieve both safety and liveness with high probability. (Whereas Nakamoto consensus does guarantee both with high probability, as long as the security parameter k is sufficiently large.)

> *Guess #4: the new trusted setup assumption (A1) breaks the proof of the PSL-FLM impossibility result.*

All the proof of the PSL-FLM impossibility result requires is for a Byzantine node to be capable of simulating the behavior of four honest nodes. Assuming that there's a genesis block that is unknown in advance does not at all interfere with such simulations.

> *Guess #5: the random oracle assumption breaks the proof of the PSL-FLM impossibility result.*

As with the previous guess, the assumed existence of a random oracle (accessible to both honest and Byzantine nodes) does nothing to prevent Byzantine nodes from simulating sets of fictitious honest nodes.

The actual explanation. The real reason is:

Byzantine nodes are strictly less powerful in our model of Nakamoto consensus than in the standard permissioned model.

If true, this would resolve the seeming contradiction between the guarantees of Nakamoto consensus and the PSL-FLM impossibility result: the former works with a weaker type of adversary and thus can tolerate a higher fraction of them than would otherwise be possible.

Remember assumption (A4'), the stronger version of (A4) satisfied specifically by the proof-of-work version of longest-chain consensus? In our proof-of-work model, a Byzantine leader is restricted to propose at most one block in its round. This restriction has no analog in the standard permissioned model, and it rules out (among other things) the canonical ploy of a Byzantine node, which would to be fabricate two conflicting blocks (both belonging to the same round), tell some honest nodes about one of them and the rest about the other one. The guarantees provided by Nakamoto consensus prove that taking away this and similar Byzantine strategies fundamentally weakens the power of Byzantine nodes, thereby turning the impossible into the possible.

To better appreciate this point, I encourage you to revisit our analysis of the permissioned version of longest-chain consensus in Lecture 8. In that lecture, the PKI assumption allowed us to easily satisfy assumptions (A2) and (A3) about the verifiability of leader selection—the leader of each round was common knowledge (e.g., derived from a shared global clock or pseudorandomness derived from the blockchain state) and, because of the PKI assumption, a round's leader and only a round's leader was in position to make (appropriately signed) block proposals in that round. This prevented Byzantine nodes from, for example, masquerading as leaders out of turn or spoofing block proposals by leaders of previous rounds. If you drop the PKI assumption, you'll find that any attempt to rework longest-chain consensus breaks down because of the enlarged space of feasible Byzantine strategies—in particular, you'll find it difficult to enforce assumption (A4), that every proposed block can only name as a predecessor a block that was first created in an earlier round. (And of course, we know from the PSL-FLM impossibility result that any such attempt must break down.)

In Nakamoto consensus, proof-of-work leader selection and assumption (A4') render these additional Byzantine strategies impossible, despite the lack of a PKI assumption. When we implement longest-chain consensus with proof-of-stake sybil-resistance in Lecture 12, Byzantine nodes will be free to make multiple block proposals in the same round and we'll need to go back to making the PKI assumption (which will in any case be quite natural in that context).

Where does the PSL-FLM proof break? The preceding discussion explains why the guarantees of Nakamoto consensus don't automatically contradict the PSL-FLM impossibility result— proof-of-work sybil-resistance effectively restricts the strategies that are otherwise available to Byzantine nodes. To complement this understanding, let's revisit the proof of the PSL-FLM impossibility result (from Lecture 3) and see how it breaks down in the proof-of-work setting.

The proof there posited a hexagon of honest nodes with inconsistent initialization files. There were three cases, with each case corresponding to a bona fide instance of Byzantine

broadcast with two honest nodes (who acts as in the hexagon) and one Byzantine node (who simulates the joint behavior of the remaining four hexagon nodes, thereby tricking the two honest nodes to behave exactly as they would in the hexagon). The three cases (along with the termination, validity, and agreement requirements) imposed a contradictory set of three conditions on the behavior of any correct protocol, thereby showing that no such protocol can exist.

In all our lectures on permissioned consensus (Lectures 2–8), we didn't worry about nodes' computational power—we assumed that honest nodes were powerful enough to carry out whatever a protocol asked of them, and that Byzantine nodes could do whatever they want (other than break cryptography). In particular, in the proof in Lecture 3, we didn't blink an eye when some Byzantine node needed to simulate a collection of four (fictitious) honest nodes from the hexagon—in that context, it would have seemed crazy to think the Byzantine nodes can't possibly do 4 times as much computational work as what the honest nodes are expected to do.

In the proof-of-work setting, however, we're making fine-grained assumption about the power of Byzantine nodes (namely, that their collective hashrate is less than the collective hashrate of the honest nodes). And simulating another node's hashrate requires having that hashrate yourself. For example, if all nodes (honest and Byzantine) have the same amount of hashrate, a Byzantine node could simulate one honest node but not four—the node can't magically fabricate four times as much hashrate as it actually has. In other words, because proof-of-work makes simulation costly, Byzantine nodes can no longer carry out the "quadruple-simulation strategies" that are needed to push through the proof of the PSL-FLM impossibility result. And as the guarantees of Nakamoto consensus show, this breakdown of the proof is fundamental and cannot be salvaged by some different and more clever argument.

6.4 Does Nakamoto Consensus Need a Shared Global Clock?

To what extent does our analysis of Nakamoto consensus require the nodes to agree upon a common notion of time? We are working in the synchronous model, which by default comes equipped with a shared global clock. But does the Nakamoto consensus ever use it? In the version we've been discussing thus far (in which the difficulty parameter τ magically falls from the sky), the answer is no: Rounds are defined in an event-driven way (as opposed to by the passage of time), so nodes must agree only on valid puzzle solutions and not necessarily on any common notion of time. This is different from all the other consensus protocols (based on longest-chain or otherwise) that we'll study, which require nodes to have (at least approximately) synchronized clocks. (For example, this assumption will be important for the proof-of-stake longest-chain protocols that we discuss in Lecture 12, which produce blocks on a deterministic schedule.)

Unfortunately, as we'll see in Section 7, Nakamoto consensus does need to rely on a notion of time to implement a "difficulty adjustment" algorithm that is responsible for tuning the difficulty parameter τ to target a particular rate of block production. Even with difficulty adjustment, however, Nakamoto consensus tends to be more forgiving of loosely synchronized

clocks than other consensus protocols (longest-chain or otherwise).

7 Difficulty Adjustment

References