Amaroo's Consensus Strategy: Ultra Terminum

by Max Kaye

Nos Ultra Terminum Ultimum Imus, Nunc

July 22$^{\text{nd}}$ 2025

Abstract

This paper explains and constructs *Amaroo's* consensus mechanism: *Ultra Terminum.*

Ultra Terminum (UT) is a proof-agnostic cross-chain consensus strategy with multiple scaling configurations, and provides a practical solution to the Blockchain Trilemma (a.k.a. Scalability Trilemma). Although UT is proof-agnostic, using *Proof of Work* as the foundational method of proof provides decisive advantages to security, scalability, and user experience.

The Trilemma takes $O(c)$ as the measure of a traditional blockchain's scalability – c being a normal computer's capacity. UT's simplest configuration, UT$_1$, scales quadratically — $O(c^2)$. UT's other scaling configurations are: UT$_2$ ($O(c^3)$), UT$_3$ ($O(c^4)$), and UT$_\aleph$ (theoretically $O(n)$, where $n \sim$ size of the network).

Proof of Reflection, a new consensus primitive, is the key to UT's performance and security. Proof of Work configurations of UT meet *or exceed* the security of traditional PoW blockchains.

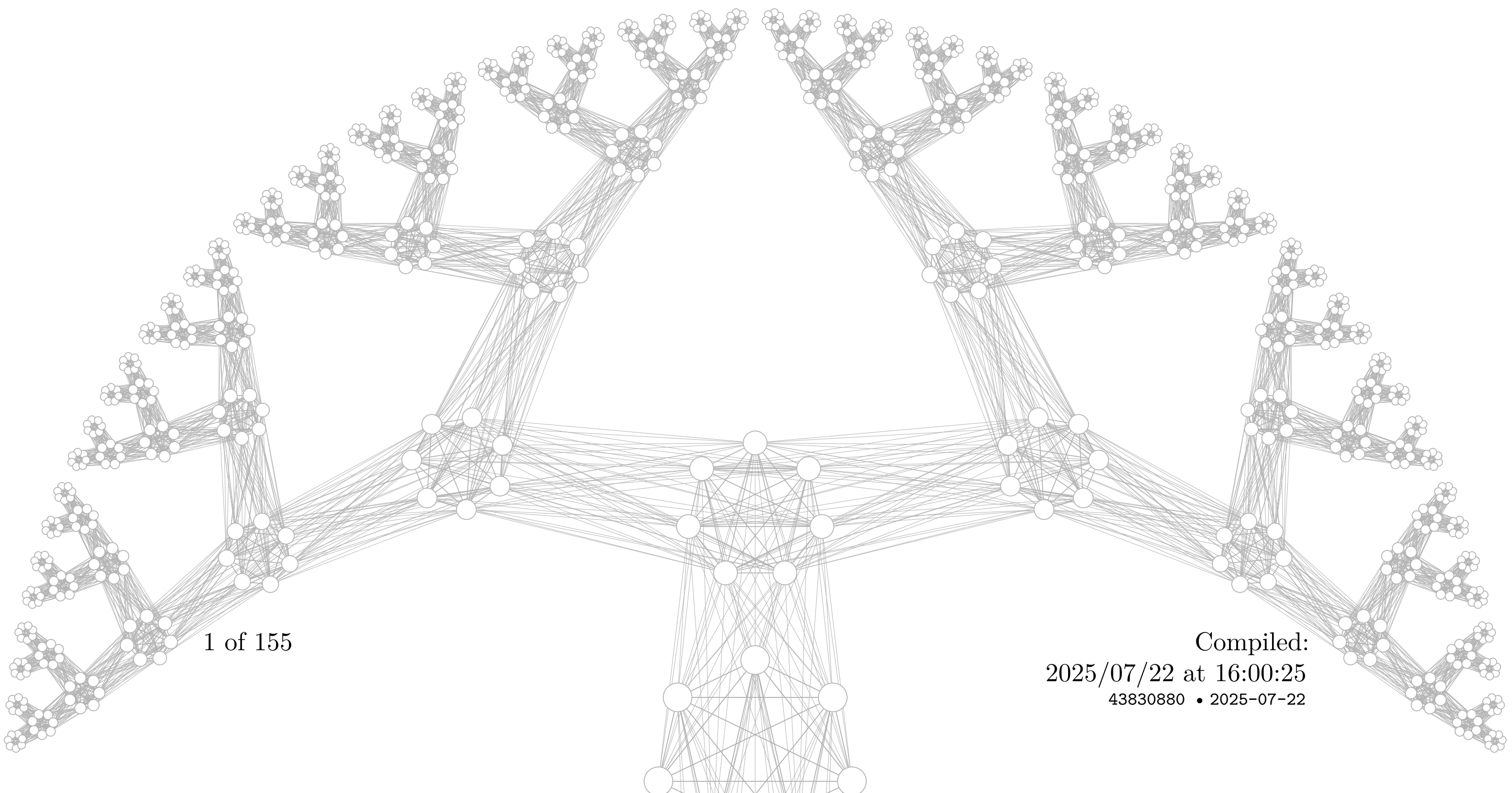
With reasonable parameters (i.e., a per-chain load comparable to Bitcoin, and one level of nested dapp-chains), a conservative estimate of capacity — using Amaroo's intended launch configuration (UT$_2$) — exceeds 350K TPS without *layer 2* scaling methods.

Capacity of UT$_\aleph$ is *independent* of those parameters and is theoretically *unbounded* by the architecture. It is combined with the previous variants to produce UT$_{\aleph 1}$, UT$_{\aleph 2}$, and UT$_{\aleph 3}$.

Confirmation times within UT are $O(c^{-1})$, i.e., confirmation rate is $O(c)$. The initial configuration will have a *maximum* confirmation frequency exceeding ∼10 Hz. Typically, sharded blockchains have a confirmation frequency < 0.2 Hz.

Some existing sharded (or otherwise $O(c^2)$) blockchain designs claim to be capable of 1M TPS. By comparison, UT$_2$ configurations — with *equivalent* parameters — have a maximum capacity exceeding 10B TPS. Such a configuration would have a system-wide maximum throughput exceeding 2000 GB/s of transaction data, and a confirmation frequency exceeding 400 Hz. UT out-scales the architectures of Eth2, Polkadot, Cardano, and Solana by orders of magnitude.

This paper's source code available at https://github.com/amaroohq/whitepaper



Contents

1 Introduction

Blockchains are complex systems and each component affects the system in a number of different ways. When designing a blockchain network, we must therefore carefully consider these components and their interactions, and the context that the chain will operate in. These choices are encoded in the protocol and the consequences of those choices can last for the entire lifetime of the network.

Naturally, when designing systems, some problems are left to be solved later – not everything can or should be done up-front. The most common problem that blockchain designers have left in the 'solve later' pile is the long-term *scalability* of the network. In turn, this neglect gave rise to a generation of blockchain designs that tried (and failed) to provide a scalable solution without abandoning the other core properties of a blockchain. Existing chains began to adapt, too – Bitcoin introduced SegWit, and Ethereum developed a plan to transition to a sharded design secured by Proof of Stake (an effort which is still ongoing, 10 years later). Despite all of this, a solution to this problem remains elusive.

Before long, this problem was codified as a trilemma by Vitalik Buterin. In this paper, we'll refer to it as the *Blockchain Trilemma* or *the Trilemma*, though it is also known as the *Scalability Trilemma*. It states that, for blockchain designs, there is a 'choose two out of three'-type relationship between (i) security, (ii) scalability and (iii) decentralization (discussed below in Section 1.1).

The shortcomings of popular existing designs which have prioritized security and decentralization are readily apparent thanks to *congestion*: transactions take longer to be confirmed, fees rise, business is disrupted, and costs explode. When security and decentralization are prioritized over scalability, there will, inevitably, be a shortage of capacity at some stage.

Other combinations of trilemma properties have been tried as well. Secure and scalable "blockchains" have been attempted, but are fragile and centralized. Scalable and decentralized systems exist, but lack the fundamental properties of a blockchain, such as presenting a single, consistent record of events to all users.

Amaroo's underlying consensus mechanism, *Ultra Terminum* (UT), is a cooperative cross-chain consensus strategy that addresses the *Trilemma* and creates cohesion between chains.

Compared to existing consensus methods, UT provides *equal or better* security properties (including Bitcoin's PoW method and variants, and *all* PoS variants). This is because UT combines existing consensus methods and, by way of construction, UT can only *add* security to these methods.

UT is capable of supporting millions of transactions per second with similar chain parameters (like block size) to those of traditional blockchains (like Bitcoin or Eth1[1]).

1.1 The Blockchain Trilemma

The trilemma is as follows:

Given c (computational resources per node, e.g., computation, bandwidth, storage, etc.), and n (the size of the network, e.g., transaction throughput, state size, user population, market cap, etc.), blockchain systems have no more than two of these three properties:

- Decentralization — the system can operate with participants that have only $O(c)$ resources (e.g., a laptop, a raspberry pi, a VPS; typical internet bandwidth, storage space, etc.).
- Security — the system is secure against attackers with up to $O(n)$ resources.[2]
- Scalability — the system can process $O(n)$ transactions, with $O(n) > O(c)$; this means that, as the network grows, the throughput of the system grows faster than the computational resources required per user.

[1]The term *Eth1* refers to the original Proof of Work Ethereum (pre-merge), and *Eth2* refers to the Sharded Proof of Stake Beacon Chain. The terms were originally adopted to describe the different designs on the Ethereum Roadmap; but they've since been deprecated. They've not been updated in this paper because they are functional for our purposes: to distinguish between PoW Ethereum and the various PoS iterations.

This definition of scalability is perhaps problematic. If the network, which has $O(n)$ TPS demand, becomes bottlenecked by an $O(c^2)$ scaling method, is the network really scalable? I prefer an alternative definition of scalability: the system can process $O(n)$ transactions in $O(1)$ time, i.e., getting a transaction confirmed is neither too expensive nor delayed as n and/or c change.[3]

Why mention $O(c^2)$ scaling particularly? Why is that an important breakpoint for scaling configurations? In a word: sharding. The standard method of sharding (or hosting child-chains generally) is to replace transactions with shard-headers in the host-chain. Extra data might also be required. If the host-chain has $O(c)$ capacity, then it should support[4] $O(c)$ shards. Each shard has $O(c)$ capacity also, thus the full system has $O(c^2)$ capacity.

Aside

Traditionally, the use of big O notation when discussing the trilemma has not conformed to the strict mathematical meaning; instead, it is used quite loosely as a way to analyze the scalability of a blockchain architecture. Naturally, we'll continue this tradition.

1.1.1 Core Conflict

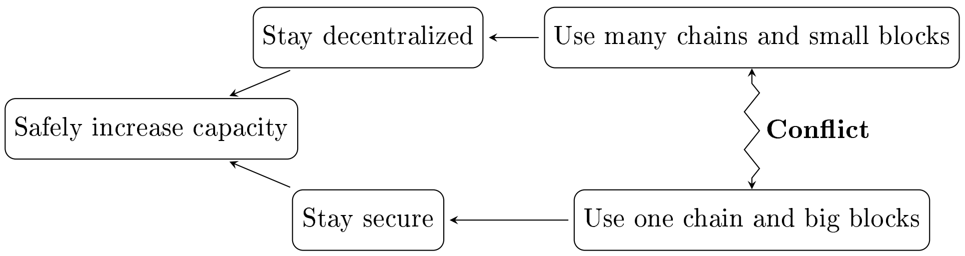


Figure 1: A "cloud" diagram of the core conflict of the *Blockchain Trilemma*. $A \to B$ can be read "A so that B"; and $A \leftarrow B$: "A requires B".

Figure 1 shows a cloud[5] for the core conflict. It reads: *safely increasing capacity* requires that we *stay decentralized*. *Staying decentralized* requires that we *use many chains and small blocks*. *Safely increasing capacity* requires that the network *stays secure*. *Staying secure* requires that we *use big blocks*. Big blocks are the opposite of small blocks, so we have a *conflict*. Additionally: using multiple chains compromises security; and big blocks compromise decentralization.

To understand the core conflict, we need to look at the underlying assumptions.

Buterin writes[6] in the Ethereum sharding FAQ regarding the *many chains with small blocks* strategy:

Quote

> [...] This greatly increases throughput, but at a cost of security: an N-factor increase in throughput using this method necessarily comes with an N-factor decrease in security, as a level of resources 1/N the size of the whole ecosystem will be sufficient to attack any individual chain. [...]

He writes, regarding the *big blocks* strategy:

[2] Here, by convention, "up to $O(n)$ resources" means that the system remains secure against attackers with less than some critical proportion of the network's block production capacity — e.g., less than 1/2 for typical PoW chains, and less than 1/3 for typical PoS chains.

[3] This does not mean there is no fee market for transactions, nor that localized congestion will not happen. Rather, we are interested in an equilibrium that provides substantial value and capacity whilst avoiding transactions being as free as sending an email.

[4] Presuming a secure method of sharding is known and in use, and $O(1)$ load per shard-header.

[5] Clouds are diagrams of explanatory implications which reveal a conflict between two (apparently) necessary things. For more on clouds, see It's Not Luck by Eli Goldratt (1994).

[6] See *On sharding blockchains FAQs*.

Quote

> [...] such an approach inevitably has its limits: if one goes too far, then nodes running on consumer hardware will drop out, the network will start to rely exclusively on a very small number of supercomputers running the blockchain, which can lead to great centralization risk.

A mistaken way to break the conflict is *merged mining* (a.k.a. AuxPoW). This method attempts to share security between chains, so that one might be able to have decentralization and security via small blocks and merged mined chains/shards (sometimes called side-chains). Buterin writes:

Quote

> [...] If all miners participate, this theoretically can increase throughput by a factor of N without compromising security. However, this also has the problem that it increases the computational and storage load on each miner by a factor of N, and so in fact this solution is simply a stealthy form of block size increase.

Figure 2 shows the cloud for the merged mining conflict.

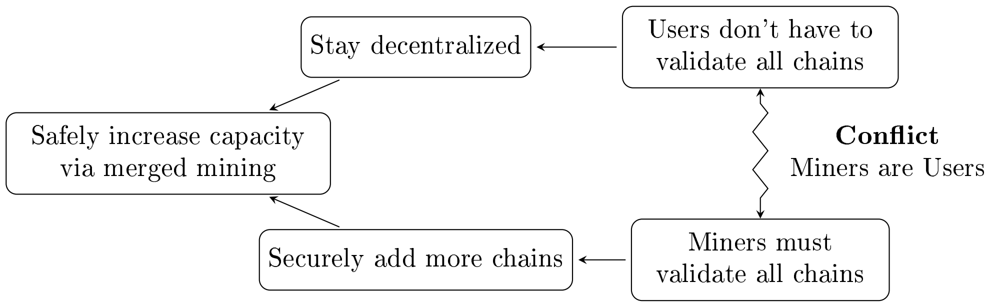

Stay decentralized
Users don't have to validate all chains
Safely increase capacity via merged mining
Conflict
Miners are Users
Securely add more chains
Miners must validate all chains

Figure 2: A cloud showing the scaling conflict of *merged mining*.

An underlying assumption here is that maximally sharing security across the network requires miners to maintain a record of all chains and do validation on all those chains. The naive solution (using many chains, mentioned above) conflicts with secure methods of merged mining. It seems like progress is impossible.

We have some hints to conditions that might belong to a solution:

- We can share security between chains/shards and can use small blocks.
- We can share security without miners keeping and validating all chains/shards.

This is the crux of the problem: how do you construct a network of blockchains (scalable) such that attacking an individual chain is about as difficult as attacking the full network (secure), whilst ensuring that the security of the network does not require validating all chains (decentralized)?

Prior Assumptions Here are some prior underlying assumptions that are either common or which I expect to be:

- Sharing PoW security requires merged mining.
- Sharing PoW security requires that chains use the same hashing algorithm.
- Multiple (non-merged-mined) PoW chains using the same hashing algorithm means that at least some of those chains are vulnerable to a 51% attack.
- Simultaneously securing a network with PoW and PoS is not possible without compromises (like that PoW miners could DoS PoS validators or vice versa).

- It is unsafe for miners/validators to build on unvalidated histories (as is done with SPV mining, which *is* unsafe).

Aside

I call them *assumptions*; some people will likely (and rightly) take issue with that and call them *conclusions* instead. For our purposes there isn't really a difference; I include them here so that I can later show you conditions under which they are all simultaneously false.

1.1.2 Conjecture: A Principle of Scaling

A scalable system *can* have components with complexities worse than $O(c)$ *if and only if* those components are not *bottlenecks*, i.e., *constraints*. As long as there is *excess capacity* in those components, the system can still scale.

1.2 Ultra Terminum

Quote

> Decoupling the underlying consensus from the state-transition has been informally proposed in private for at least two years—Max Kaye was a proponent of such a strategy during the very early days of Ethereum.
>
> — *Dr. Gavin Wood; Polkadot Whitepaper (2016), s2.2*

In essence, *Ultra Terminum* (UT) is not a method to scale a single blockchain, rather a method to scale a *network* of blockchains.

UT differs from existing consensus methods[7] in that it is a *novel modification* to particular components of preexisting consensus algorithms. These modifications allow those consensus algorithms to cooperate. This improves the maximal security and decentralization of the network, whilst also providing a foundation for scalability.

At the core of *Ultra Terminum* is a new method for sharing security: *Proof of Reflection* (Section 2). This technique (which works *in conjunction* with PoW, PoS, etc) allows the incremental construction of complex blockchain networks with powerful scaling properties. Those blockchains use the technique *Proof of Reflection* to *mutually secure each other*, forming something called *the simplex* (Section 3.2). Blockchains at this level are called *simplex-chains* and are able to host *dapp-chains* — chains that are dedicated to a dapp, and they can be application specific (e.g., a DEX) or general (e.g., a chain hosting an instance of the EVM). The vital properties provided by UT allow heterogenous chains to form a cohesive network and cooperate in securing each other. *Proof of Reflection* is how *Ultra Terminum* (excluding nested chains) scales with order $O(c^2)$, and is the basis for unbounded $O(n)$ scaling. UT's higher-order scaling configurations, $O(c^3)$ and $O(c^4)$, require dapp-chains: chains that are application-specific and which inherit security properties from the foundational structure. As UT is primarily a method of *structuring* a blockchain network, the scalability configurations mentioned herein *do not include* ***layer 2*** *methods.* That means that *layer 2* techniques (e.g., state/payment channels, ZK/optimistic rollups) can be implemented *on top* of UT.

UT's novel structure means that confirmation times within UT are *sub-constant* and of order $O(c^{-1})$ (i.e., the confirmation rate is $O(c)$). This means that, as the network grows and c increases, confirmation times in UT will approach 0. This is an improvement over existing architectures, which are, ideally and at best, $O(1)$.

Practically, there is little difference between a single blockchain and the simplex to the user. *Proof of Reflection* allows each simplex-chain, and each hosted dapp-chain, to do efficient SPV proofs regarding state-entries on other simplex-chains or dapp-chains. This allows for the creation of

[7]Whether *Ultra Terminum* is a consensus *method* or not is not immediately clear. On the one hand: a new combination of methods is still a method. On the other hand: UT provides a way to modify other consensus methods via the fork rule, and UT doesn't provide a way to run a *single* blockchain. This is why I introduced it as a cross-chain consensus *strategy*.

rich cross-chain functionality, such as the ability to use a single network-level token across all simplex-chains and dapp-chains.

Although SPV proofs can be hundreds of bytes long, UT is structured so that their length does not significantly impact scalability. UT does this by prioritizing a clean and simple protocol for simplex-chains, and allowing dapp-chains the freedom to implement whatever state- and transaction-protocols are suitable for the application in question. This means that complex or inefficient data-structures for a dapp-chain's state do not impact the network globally, preventing bottlenecks in foundational components.

Finally, a UT simplex can be modified to enable a *tiling of simplexes* (Section 6). This technique *removes the upper bound* on a UT network's growth, without sacrificing security of the network.

1.3 Paper Roadmap

The main goals of this paper are to document, explain, discuss, and evaluate UT and its components. Some secondary goals of this paper are to be *simple and clear* — this is why the paper is written in this voice. When I say "we", I am referring to *you and I*. This document was written to be read.

Our journey is structured thus:

- Section 2 constructs the main primitive, *Proof of Reflection*, and covers related topics such as *conversion of work*.
- Section 3 uses *Proof of Reflection* to construct the *Simplex*, *Ultra Terminum* (UT), and its scaling configurations: UT_1, UT_2 and UT_3.
- Section 4 covers the practical considerations for UT: the availability of blocks, state transition, protocol variants, confirmation times, the PoR graph, block-DAGs, DoS attacks, NIPoPoWs, simplex security, cross-chain transactions, expedited transactions, and a possible initial configuration.
- Section 5 analyzes the scalability of UT.
- Section 6 constructs a *Tiling of Simplexes* and $UT_\aleph$, scaling horizontally with $O(n)$ capacity.
- Section 7 briefly summarizes UT's resistance to attacks.
- Section 8 concludes by evaluating all UT variants against the Trilemma.
- Section 9 lists the major criticisms of UT with commentary.

The appendices contain comparisons to other networks, details on the iterations of the simulation (Section 4.10.4), and nonessential protocol analysis.

2 Proof of Reflection

In this section we will build up to and discuss a new consensus primitive: *Proof of Reflection* (PoR). This is a proof that one blockchain's headers have been confirmed by a second chain. Using these proofs, we can share security between these two chains, such that attacking either chain is as difficult as attacking both.

Our starting point is the question: is there anything *in principle* that prohibits blockchains sharing security? We already have some examples of this (e.g., merged mining), but those existing methods require each miner to opt-in and aren't general. Can we come up with a general way to share security between chains?

Aside

A note on the term "miner": consensus protocols sometimes choose a new name for the role of *block producer*. For example: "validator", "baker", "collator", etc. In this document, the term "miner" usually refers to the generic role of *block producer* in an inclusive sense, rather than specifically to block producers of PoW based-chains.

Our starting point is the idea of one chain 'tracking' another chain via its headers — it's the sort of thing that would enable a smart contract to do on-chain SPV proof evaluation. This isn't a new idea: in 2013, I proposed a system which used this method to support rich cross-chain exchange.[8] I also wrote a simplified implementation of this method for Bitcoin SPV proofs in the very early days of Ethereum[9] — a precursor to the later-successful BTC Relay[10] (although BTC Relay seems to have been abandoned in late 2017).

What should we call an on-chain version of a headers-only chain? Since there is no established term, let's call it a "projection", and let's call the *act* of one chain creating a projection of another "imaging".

Term

Projection: A *projection* of a chain is its *headers-only* version that has been recorded and evaluated *by a different chain*. For example, BTC Relay is a smart contract by which Ethereum previously hosted a *projection* of Bitcoin. The *act* of one chain creating and maintaining the projection of another is called *imaging*.

2.1 A Projection of Bitcoin in Ethereum

It is well understood that an Ethereum smart contract (SC) can *image* Bitcoin. This results in a *projection* of Bitcoin that is available to other Ethereum SCs (e.g., to use as the basis for SPV proofs). Since Bitcoin's consensus and PoW algorithm is simple and has low overhead, implementing the necessary logic as an Ethereum SC is viable. In principle, any chain with a headers-only mode can be imaged in this way.[11]

Let's assume that a smart contract like this exists on the Ethereum chain and it is operational.

We're going to use some tables and diagrams as we go to show the sequence of events that occur and what data is produced and/or recorded. Table 1 shows data and events for both Bitcoin and Ethereum as Bitcoin blocks are produced and the projection of Bitcoin in Ethereum is maintained. Figure 3 illustrates this. Note: Figure 3 includes some variance in Ethereum's block production rate, similar to what might be observed in a real-world environment, but Table 1 does not.

After a Bitcoin block (BTC_k) is produced, a user (it doesn't matter who) produces a transaction informing the SC of that block's header. Then, an Ethereum miner includes that transaction in a block, which updates the SC. This process repeats when the next Bitcoin block is mined, and so on.

[8]see https://bitcointalk.org/index.php?topic=198032.0, and https://bitcointalk.org/index.php?topic=598784.0
[9]https://github.com/XertroV/coppr/blob/master/chainheaders.py [m1]
[10]https://github.com/ethereum/btcrelay
[11]Practically, there can be constraints that prevent this. For example: Ethereum typically doesn't support memory hard hashes, so hosting projections of chains like Litecoin on Ethereum is potentially impossible.

Why would a chain want[12] to include a projection of another chain? The typical answer is to enable proofs of some state or that some transactions occurred on the imaged chain. This enables more complex SCs, e.g., a trustless BTC ↔ ETH market.

Table 1: (Hypothetical) Data and events for both Bitcoin and Ethereum as a projection of Bitcoin in Ethereum is maintained via Bitcoin headers being tracked by an Ethereum SC.

Time step (~15 s increments)	Bitcoin block mined	Eth block mined	Eth block contents	Eth state
⋮				
0	k			
1		j	BTC_k header	Records $\text{BTC}_{0\cdots k}$
⋮				
40	$k+1$			
41		$j+40$	BTC_{k+1} header	Records $\text{BTC}_{0\cdots k+1}$
⋮				

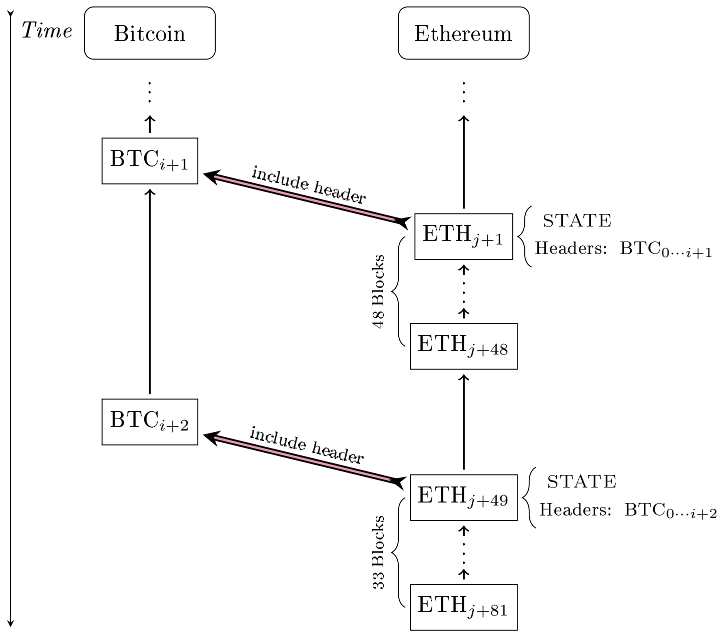


Figure 3: (Hypothetical) Bitcoin headers, as they are produced, are included in Ethereum's state via a smart contract and user made transactions. This is roughly how BTC Relay worked.

2.2 Incrementally Implementing *Proof of Reflection*

Let's consider two hypothetical and distinct blockchains — chain L and chain R. To begin with, you can imagine these as Bitcoin and Ethereum (before migrating to PoS). However, the changes that are required by *Proof of Reflection* would require major changes to an existing chain's consensus mechanism, so it's unlikely that changes like these would ever be integrated.

Our starting case for L and R is that both use Nakamoto consensus with the same Proof of Work algorithm. For simplicity, both chains also have identical block times and we won't consider block production variance.

[12]Obviously, blockchains don't have desires. We'll personify them as a short-cut, which here means approximately: the users / community / developers of a blockchain have some common goal or incentive to do a particular thing.

2.2.1 Step 1. Chain R Images Chain L

This is similar to Ethereum imaging Bitcoin — see Figure 4. (We'll omit the table this time since it's basically the same as Table 1.)

Like before, R will include L's headers as they are produced. Unlike before, let's assume this is a *protocol-level* implementation; so R miners can include L headers directly with no transaction fees.

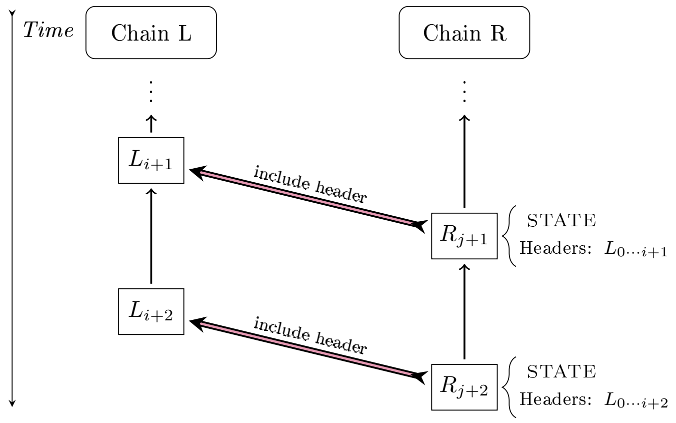


Figure 4: Step 1. Chain R images Chain L; thus Chain R hosts a *projection* of Chain L.

2.2.2 Step 2. L Images R

Similar to R, L will begin including R headers in each block via appropriate protocol-level changes. Just as R hosts a projection of L, L now hosts a projection of R. This is shown in Figure 5 and Table 2.

Table 2: Step 2. Both Chain L and Chain R host a projection of each other.

Time	L block made	L block contents	L state	R block made	R block contents	R state
⋮						
0	k	R_{j-1} header	Records $R_{0\cdots j-1}$			
1				j	L_k header	Records $L_{0\cdots k}$
2	$k+1$	R_j header	Records $R_{0\cdots j}$			
3				$j+1$	L_{k+1} header	Records $L_{0\cdots k+1}$
⋮						

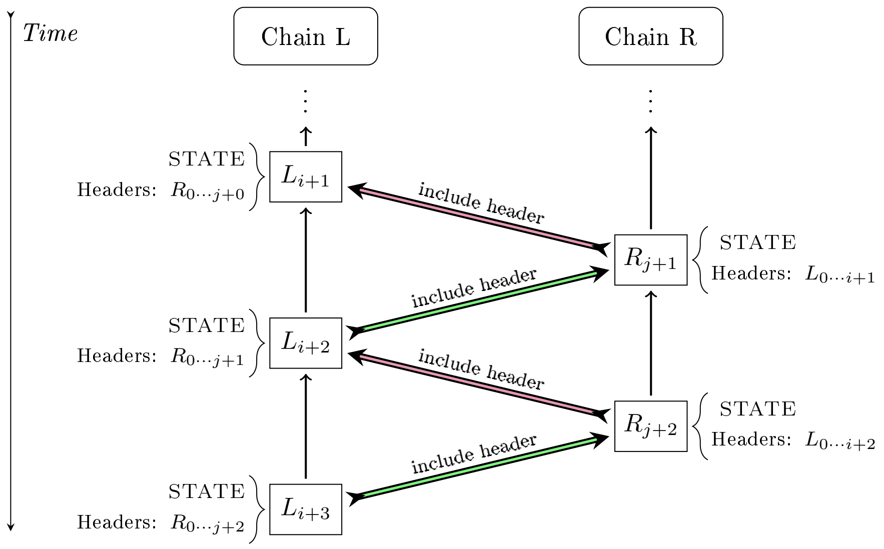
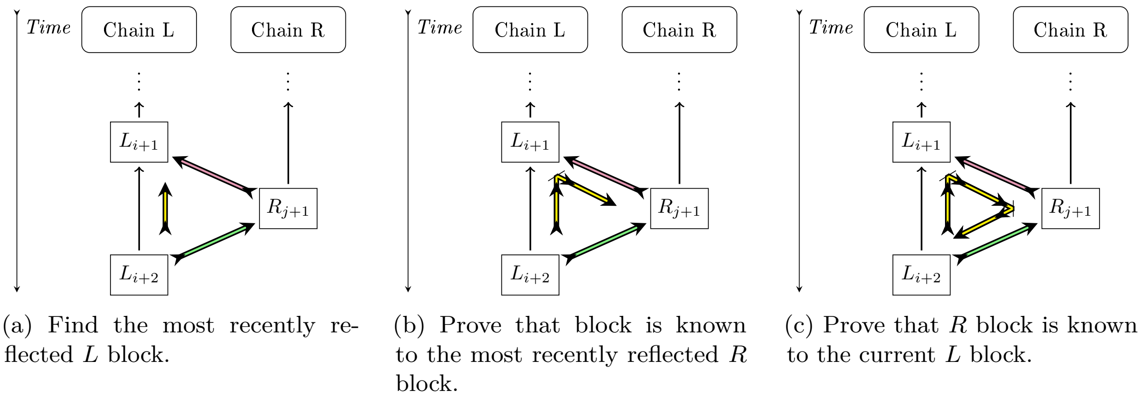


Figure 5: Step 2. Chain L and Chain R contain a projection of each other's headers-only chain.

2.2.3 Step 3. A Reflection of L in R

Can we use a projection of a chain for a different purpose? What happens if Chain L tracks whether Chain L's history is confirmed within Chain R? This can be done via merkle branches[13] that prove Chain R's relevant state. In essence, Chain L uses its projection of Chain R to prove that its *own history* matches that of its *projection* in Chain R. Chain L proves that it is *reflected* in Chain R.

What does this proof look like? The following progression is shown in Figure 6. First, Chain L must prove that its history is reflected, so we first find the most recently reflected header, L_{i+1} (ideally, this is the previous L block). Secondly, we want to prove that L_{i+1} is also the *best block* (for Chain L) according to *Chain R's* projection of Chain L, using the best known R block, R_{j+1}. For that, we need a merkle branch showing L_{i+1} is part of R_{j+1}'s state — this is sometimes referred to (in this paper) as the *missing* merkle branch. Thirdly, we want to prove that R_{j+1} is the *best block* according to Chain L's projection of Chain R. We can do that via a merkle branch, too, but full nodes of Chain L already know whether R_{j+1} is the best block or not, so this branch doesn't need to be explicit. However, L's nodes must be able to generate it. The *full* collection of information required to prove reflection is called a *Proof of Reflection* (PoR).



(a) Find the most recently reflected L block.

(b) Prove that block is known to the most recently reflected R block.

(c) Prove that R block is known to the current L block.

Figure 6: Incrementally constructing a *proof of reflection* (PoR).

Segments of Chain L and R (events and data) are shown in Table 3 and Figure 7.

[13]Vector commitments (or verkle branches) can be used, too (this applies to most uses of merkle trees / branches in this paper). For the sake of convenience and simplicity, verkle trees won't be explicitly mentioned as an alternative unless there is a specific purpose.

Chain L now knows *which L blocks are recorded by Chain R*, i.e., which local blocks are known about by some external source. Put another way: Chain L's history is confirmed *not only* by new Chain L blocks, *but also* by Chain R blocks.

Aside **Important:** Soon, these confirmations will have real and useful meaning. Under the right conditions, an appropriate configuration of *Proof of Reflection* results in an increase in the *rate* that confirmations are acquired. This is the first hint of $O(c^{-1})$ confirmation time.

At this point, if an attacker was to publish an alternate, better Chain L history, then Chain L nodes would reorganize around the *new* history published by the attacker, and the attacker's block headers would end up being recorded in Chain R and causing a reorganization there, too. Currently, this configuration does not add any security to Chain L.

Could we use Chain L's knowledge *that its own history is reflected in Chain R* to *prevent* such an attack?

Table 3: Step 3. Chain L records which of its headers are known about by Chain R. That is: Chain L includes *proofs of reflection.* Note: "Headers" is abbreviated to "Hdrs".

Time	L block mined	L block contents	L state	R block mined	R block contents	R state
⋮						
0	k	R_{j-1} header, L_{k-1} PoR	Hdrs: $R_{0\cdots j-1}$, PoRs: $L_{0\cdots k-1}$			
1				j	L_k header	Hdrs: $L_{0\cdots k}$
2	$k+1$	R_j header, L_k PoR	Hdrs: $R_{0\cdots j}$, PoRs: $L_{0\cdots k}$			
3				$j+1$	L_{k+1} header	Hdrs: $L_{0\cdots k+1}$
⋮						

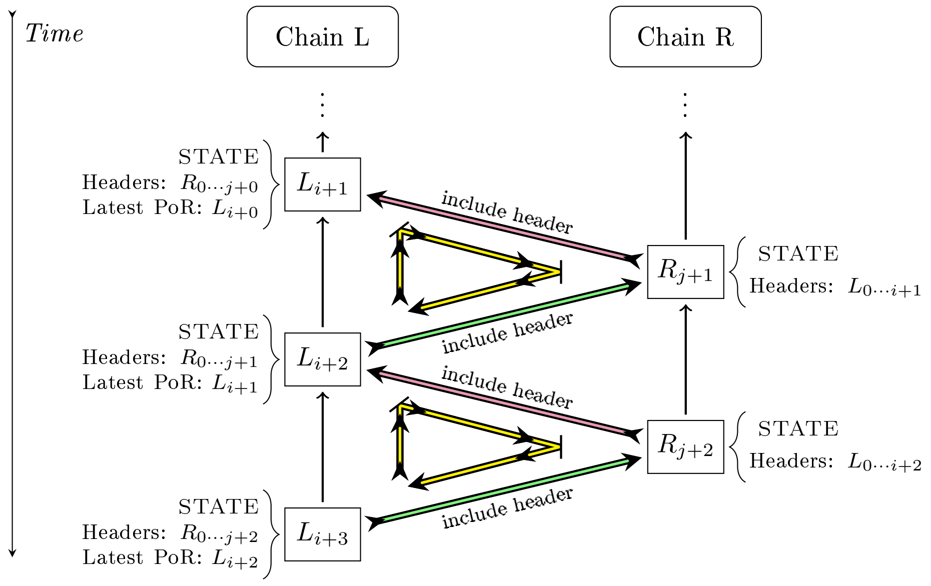


Figure 7: Step 3. Chain L includes *proofs of reflection* (PoRs) along with headers. Proofs of Reflection allow Chain L to know which of its own blocks are known to Chain R.

2.2.4 Step 4. One Way Reflection

Before we discuss a change that Chain L could make, it is important to note that chain-work done with one hashing algorithm is *not generally convertible* to ‘equivalent’ work done via another hashing algorithm. For example, there is no meaningful *generic* answer to the question *how many double SHA256*[14] *hashes is one Ethash hash worth?*

For the purposes of our hypothetical construction, let’s say that L and R do *equal work over equal time*. In the current example, that means that the work required to produce either L_i or R_j is the same. *For the sake of this construction, we’ll also presume this relationship doesn’t change over time.* Our constant of conversion is thus: 1 *R Blocks per L Block*.

Aside

> We’re not that concerned with whether this is a reasonable assumption in the real world or not; right now, we just need a way to convert the work done on each chain into the same units. Methods for converting work are discussed in Section 2.3.

Currently, the Chain L network chooses the “heaviest” (most worked) chain as its common history. Chain L calculates the “weight” of blocks (i.e., how much work went in to them) via an estimation of how many hashes were required — e.g., some number of *double SHA256 hashes*. For the purposes of illustration, let’s convert this number to be in terms of *L Blocks* instead of *double SHA256 hashes*. That’s easy, since each block is worth 1 *L Block* by definition. We can also measure the work of an L block in terms of *R Blocks* (1 *L-Block* = 1 *R-Block* by the constant of conversion above).

How can the network choose the heaviest chain? Well, a traditional blockchain might use a simple recursive function like Algorithm 1.

Algorithm 1 Vanilla chain-weight algorithm (typical of traditional blockchains).

```
procedure ChainWeight($blocks$, $state$)                         ▷ The weight of a chain
    if length($blocks$) == 0 then
        return 0
    end if
    return BlockWeight(head($blocks$), $state$) + ChainWeight(tail($blocks$), $state$)
end procedure

procedure BlockWeight($block$, $state$)                ▷ The weight of an individual block
    return LocalBlockWeight($block$, $state$)        ▷ NB: defined by the consensus method
end procedure
```

Now that we can convert block weights between L blocks and R blocks, could L’s ChainWeight algorithm incorporate the idea that Chain R had confirmed part of L’s history? Could Chain L use this to thwart some types of attack?

Yes, and we must modify the block-weight calculation so that it accounts for work contributed by Chain R. Algorithm 2 is such an algorithm. Essentially, additional weight is added to a block when it is *the best block* known to Chain R, i.e., when, according to Chain R, it is at the tip of Chain L. Note that this weight is still added if Chain R knows of multiple competing chain-tips.

What is the meaning and impact of this change?

The *meaning* of this change is that Chain L now incorporates work done on Chain R *into Chain L’s own calculation of the heaviest worked chain.*

When a chain does this we say *Chain L (or Chain L’s work) is **reflected** in Chain R.* This technique is what is meant by the term *Proof of Reflection*.

[14]Bitcoin uses $\text{Hash}(x) = \text{SHA256}(\text{SHA256}(x))$ as its PoW hash.

Algorithm 2 Modified chain-weight algorithm to factor in reflections.

```
procedure ChainWeight(blocks, state)                                  ▷ The weight of a blockchain
    if length(blocks) == 0 then
        return 0
    end if
    return BlockWeight(head(blocks), state) + ChainWeight(tail(blocks), state)
end procedure

procedure BlockWeight(block, state)
    $W_l$ ← LocalBlockWeight(block)                    ▷ Weight due to work done producing block
    $W_r$ ← ReflectedBlockWeight(block, state)                      ▷ Weight added via reflection
    return $W_l + W_r$
end procedure

procedure ReflectedBlockWeight($L_i$, state)
    $w \leftarrow 0$                                                              ▷ Sum of weights
    $R_{\text{blocks}}$ ← ReflectingRBlocks(state)          ▷ Chain R blocks that reflect the local chain
    for $R_i$ in $R_{\text{blocks}}$ do
        RCHs ← ReflectedChainHeads($R_i$, state)                 ▷ Local blocks reflected by $R_i$
        if $L_i$ in RCHs then
            $w \leftarrow w$ + WeightOf($R_i$, state)            ▷ NB: network specific, e.g., Algorithm 3
        end if
    end for
    return w
end procedure
```

Term

Proof of Reflection (PoR): The consensus technique whereby a blockchain becomes more difficult to attack by including work done by reflecting blockchains in its *fork rule.*

One particular *impact* of this change is that a doublespend attack on L (e.g., withholding a privately mined chain-segment that reverts a transaction) must now be performed *not only* against Chain L, *but also and simultaneously* against Chain R.

Why? The attacker's privately mined L blocks *are not known about* by Chain R. Rather, Chain R knows about the *public* Chain L history *against which the attack competes.* Thus, *either*:

- the private chain-segment must contribute more total work to the Chain L blockchain than the public chain-segment does (*including* the relevant Chain R chain-segment); *or*
- the attacker must *additionally* produce a private Chain R chain-segment such that the *total* work of both private chain-segments is greater than the total work of both public chain-segments, and publish both chain-segments simultaneously.

Note that, at this point, there is no benefit to Chain R 's security. That's because Chain R isn't 'reading' the reflected work back from Chain L. Thus a doublespend attack against Chain R has the expected, non-reflected profile — it isn't more difficult to attack Chain R yet. However, Chain R can take advantage of the reflection. The main requirements are: the inclusion of appropriate proofs of reflection that show known Chain R blocks according to Chain L, and an update to Chain R's block-weight calculations to account for the reflected work. *Proof of Reflection* doesn't automatically secure both chains; each chain can proactively and independently take advantage of *Proof of Reflection.*

Naturally, if there were a large difference in target block frequencies (e.g., 10 minutes vs 15 seconds) then there would also be a good deal of latency between the points where the higher-frequency chain gains the security benefit from reflected work. For this reason, *Proof of Reflection* is most useful between high frequency chains, or chains of similar frequencies. One downside of this is that shortening the block production frequency requires the inclusion of more block headers. In the scheme of things, this can be somewhat significant but it is not a deal-breaker.

Exactly how one chain can properly account for reflected work requires that we cover how to compare (and convert) that work, and is the topic of Section 2.3.

Note that, as the Chain L tip is gaining reflections from Chain R, miners on Chain L are incented to include as many novel Chain R headers and PoRs as possible. That's because each new Chain R header (with a PoR) will increase the weight of the *ancestors* of the Chain L draft block, which helps the draft block compete with other draft L blocks. This increases the overall chain-weight that the miner is building on, and thus contributes to their block becoming part of the most-worked chain.

Aside

Where do Chain L miners get PoRs from? There are multiple answers, but one is for miners of Chain L to request them from Chain R nodes — light-client protocols often support this sort of thing. The problem is discussed in Section 4. For now, it's okay to assume that PoRs are broadcast alongside headers.

There are still potential attacks on Chain L. For example: what if an attacker mines a doublespend in private and produces a longer chain-segment than the honest chain? That is, the attacker's segment — *excluding reflections* — is heavier than the honest chain-segment. At this point the attacker can publish their blocks even though the honest chain-segment — *including reflections* — is heavier. Chain L nodes would *not* reorganize around this new chain-segment, so why would an attacker do this? If the projection of Chain L in Chain R *does not account for reflections*, then the attacker's chain-segment will appear (to Chain R) to have more work than the honest chain-segment. Thus the *projection* of L in R will reorganize to favor the attacker's chain-segment. If the attacker has more hash power than the honest miners (i.e., $q > p$[15]) then they might[16] be able to use this reorganization as a foothold — either to launch a traditional 51% attack against L, or to attack SPV verification and light clients.

Aside

Note: Chain R is not *required* to actually evaluate Chain L's tip. PoR can still work if Chain R simply records every Chain L header that it can, and nothing more. However, this increases the complexity of a PoR implementation, and Chain R users won't have protocol-level access to a projection of L. Since a correctly-evaluated projection of L is useful (for users of either chain), we should solve this problem if we can.

How can we prevent this kind of attack? The attack is only possible because Chain R was *not* accounting for reflected weight — if Chain R's projection of Chain L accounts for reflections, then this attack is not possible. If Chain R users were *required* to run full nodes for both L and R, then we've essentially just combined L and R into one big, overly-complex blockchain — this change would thwart the attack, but it isn't a solution. Instead, we need to ensure that Chain R can cheaply and reliably evaluate the weight of L's reflections.

Let's add a field to L's header: the total chain-weight,[17] *including reflections*, of that block. If this value is *always* reliable, then it's trivial to correctly construct L's headers-only chain. With traditional blockchains (like Bitcoin) it's easy to verify the weight of a header, and thus a headers-only chain, because the header's difficulty is *already available* as part of the PoW's payload. In *this* case, though, *additional* data is required — specifically, the proofs of reflection. Full nodes of Chain L already verify that the claimed chain-weight is accurate — all the required data is contained in L

[15] In Satoshi's Bitcoin: A Peer-to-Peer Electronic Cash System the parameters p and q represent the probability that the next block will be found by an honest node or the attacker, respectively. This convention has been continued in subsequent analysis, e.g., Rosenfeld's Analysis of hashrate-based double-spending (Meni Rosenfeld; 2012), and is continued here, also.

[16] In a limited case like this, where there are only two chains, there are many options for preventing these sorts of attacks on full nodes. However, in a more general case, where there might be many reflecting chains, we need to deal with the *root cause* of the vulnerability.

[17] Instead of the total chain-weight, the sum of reflected weight works too (these are essentially equivalent).

blocks — but this doesn't help light clients. We need additional protocol changes to ensure that the *claimed* chain-weight of a header is *always* reliable.

Aside One solution is to adopt a design that allows R nodes to independently calculate or verify L's reflections without evaluating L's state. Provided that R nodes can calculate any missing merkle branches on demand, this will work. This method has substantial advantages, however, some configurations have considerable overhead. It is discussed in Section 4.3 and Section 4.4, and analyzed in Section 5.8.

We *can*, at least, guarantee that a fraud proof will *always* be possible when a malicious L block lies about its total chain-weight. Additionally, other L miners can detect the lie and link back to the malicious block as an invalid parent alongside the fraud proof. These are useful features, but they're overkill at this point.

For now, let's assume that R records L headers *and* the corresponding PoRs, and that R nodes verify L headers' chain weights.

2.2.5 Step 5. Mutual Reflection

The final step in this progression is *mutual reflection* — where both chains image one-another and include the necessary PoRs and modifications to their chain-weight algorithms. This is shown in Figure 8.

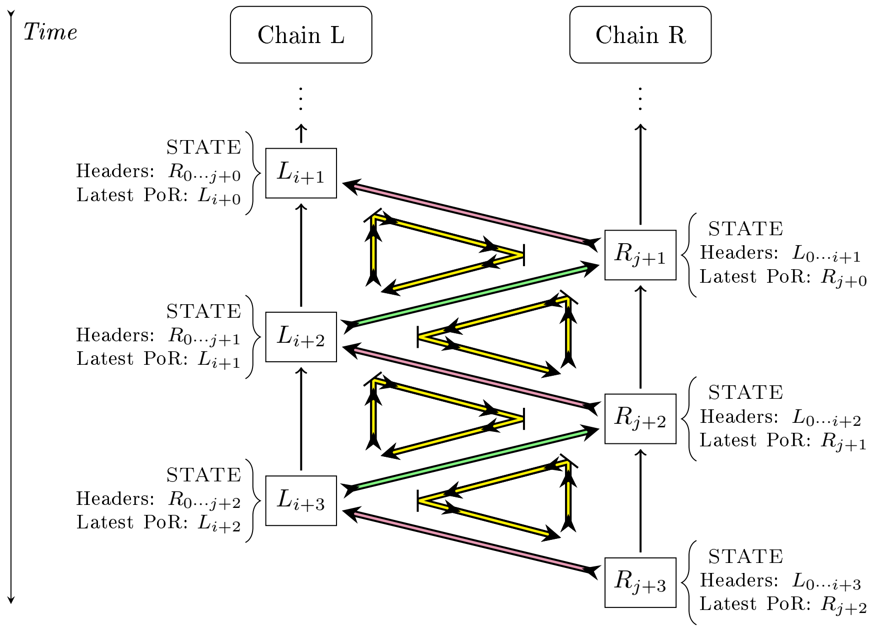


Figure 8: Step 5. *Proof of Reflection* between two UT Chains, Chain L and Chain R

When two chains (Chain L and Chain R) mutually reflect each other, detecting attacks becomes easier. The security of both Chain L and R are partially dependent on each others' histories (along with their own, of course). If one chain is attacked, where some alternate chain-segment is published, then that chain's nodes will know that those blocks have not been reflected — potentially indicating that the recently-published chain-segment was constructed in private or constructed after the fact.

There are several details that still require discussion, though, such as: *how exactly is weight contributed by a reflecting chain converted to weight in the local chain?* (discussed in Section 2.3); and *how can proofs of reflection be calculated without the requirement that miners are full nodes of both chains?* (discussed in Section 4). This last question is particularly important for moving beyond mutual reflection between only two chains.

The *essence* of *Proof of Reflection* should now be apparent. *In principle*, we can make blockchains more difficult to attack based on the idea that *blockchains can include a projection of the history of other blockchains (and confirm a chain's history like they do transactions)*. *In principle*, it is possible to increase the security of a blockchain via *reflection* and to increase the security of multiple blockchains via *mutual reflection*.

Remarks Typically, it is insecure for a weaker PoW blockchain to use the same hashing algorithm as a more popular, more heavily mined PoW blockchain. In such a situation, a small proportion of miners on the more popular chain could temporarily divert efforts to perform a doublespend or empty block DoS on the other chain, and thus the weaker chain is plainly insecure. However, if those two chains were using *mutual PoR*, then this kind of attack becomes impossible, and we've found a way for two PoW chains to coexist using the same PoW algorithm.

2.3 Comparing Incomparable Proofs of Work

For *Proof of Reflection* to work, we must alter the *fork rule* (which compares the chain-weight of two candidate best-blocks). We sum block-weights to get the chain-weight. PoW chains use *the expected number of hashes to produce a block* as the measure of *work*.[18] So, for mutually reflecting PoW chains, we need a way to *compare and convert* hashes done on R to hashes done on L (and vice versa).

If we are to convert work between PoW chains, then we *must* be able to convert between L-hashes and R-hashes *in a consistent way*. Consistency, here, means the conversion shouldn't change the outcome of the fork rule; if the fork rule says $L_{i\cdots j,a} > L_{i\cdots j,b}$ in terms of L-hashes, then it should give the same result for *all* cases when the units are changed to R-hashes.

Since both weight and hashes can be *summed*, an easy way for this to work is for us to use a *linear* conversion method (i.e., the units are *proportional*, we have some *constant of conversion* for that specific situation, and all units *share an origin*). In effect, for any two values, the *ratio* between linearly converted values is *constant* provided those values are in same units, but independent of *which* units those values are represented in. This doesn't imply that all possible *mid-states* are linearly convertible, though. However, if some mid-states *are* linearly convertible (to and from hashes), then *the fork rule should work for those units, too.*

When it comes to comparing work, we'll need to work with particular factors (or properties) that are intrinsic to all blockchains (though the particular values vary by chain). These are the factors which *naturally come as rates*, even though they're often measured in absolutes. For example: instead of measuring work in raw hashes, we'll use *hashes per block*; instead of measuring *block rewards*, we'll measure *coins per block* and the *inflation rate*. Even though the act of block production is *discrete*, blockchains have a *block target time*; a way of keeping things consistent *over time* (at least in the short term). Since a block represents a discrete amount of work, and has a known, measurable target time, we can sometimes unify the two via the idea of *rate of work*, e.g., *hashes per second*. Making these sorts of conversions up-front (where appropriate) will simplify the work to come, since we'll be working with ratios of units which naturally allow for useful conversion.

Aside When we can measure the *rate of work* in some direct way (e.g., number of hashes per block), we can also *scale block rewards according to both the network difficulty and the demonstrated rate of work*. This enables dynamic block production (i.e., blocks with a flexible minimum difficulty). This would (probably) be disruptive to a traditional blockchain, due to many stale blocks. For UT, it becomes relevant with Section 4.8, and particularly with Section 4.8.3. UT does not require support for dynamic block production, but it's useful to have as a technique in reserve (perhaps particularly for Section 3.4).

Here's the problem: *universal and generic* conversion between qualitatively different units is

[18]Alternatively, something that's *linearly convertible* to number-of-hashes (like difficulty) can be used instead.

impossible.[19] We need a specific *goal* and *context* to successfully convert — in fact, the *goal* is what *enables* us to judge whether it's a successful method of conversion or not.

We only know one specific thing at the moment: our goal is to *convert* chain-work so that *PoR works*. The *conversion* shouldn't introduce any vulnerabilities, and it should work within the constraints that we've discussed up to this point.

In order to discuss *conversion methods* we need to know the *context* within which the conversion happens. We already know some background context, like that we're using blockchains, that root tokens (which are exact and fungible) are involved via block rewards, that nodes have access to certain information via some proof or because it's already in their state, etc. This background context isn't enough, though. What specific contexts can *facilitate* conversion? This section has two examples — Section 2.4.1 and Section 2.4.2.

Before we discuss the examples themselves, let's discuss some intermediate conversions that we'll use in those two examples.

2.3.1 Theoretical Conversion

Consider a traditional blockchain (like Bitcoin, or Eth1). We know that traditional blockchains have properties specific to their blocks, like: reward per block (coins/block); a block target time (seconds/block) — or block frequency (blocks/second); and a difficulty (hashes/block). There are also *network-wide* properties, too, like the *inflation rate* (coins/second). The *instantaneous* relationship between these properties is mediated by various protocols — these protocols (e.g., difficulty adjustment algorithms) are part of the *context* of those properties and relationships. How can we use these relationships to our advantage?

Aside

With regards to Proof of Reflection, consider that we *only* need to convert *simultaneous* work. That means: PoR does not need to be able to convert chain-work between chains *over time*, only *for some given moment.*[a]

[a]At this point, we are discussing how PoR works with Proof of Work, discussions of where other methods, such as Proof of Stake, fit into the ecosystem will be discussed in Section 2.5.

The units that we have to work with are: blocks, seconds, hashes, and coins[20]. There are actually multiple types of blocks (L-blocks and R-blocks), coins (L-coins and R-coins), and hashes (L-hashes and R-hashes). We can't combine those unless we're able to convert those values to common units.

If we ignore some of the normal constraints on consensus algorithms — like where information comes from — what information could help us convert? If we had *an exchange rate* between L-coins and R-coins, then we can trivially convert between them. If we have that, then, for our current purpose, we can treat L-coins and R-coins as interchangeable units — because we can *always* convert between them. So now we have L-blocks, R-blocks, coins, and L-hashes and R-hashes.

Let's consider Chain L, and give some of these properties variables: L_f (L-blocks/s) for block frequency, L_r (L-coins/L-block) for the block reward, and L_d (L-hashes/L-block) — the difficulty. We can multiply combinations of these to get new units: $L_f \cdot L_d$ gives us L-hashes/s; $L_f \cdot L_r$ gives L-coins/s, and ${}^{L_d}/_{L_r}$ gives us **L-hashes/L-coin**.

Now, let's add that exchange rate: $X_{R \to L}$ (L-coins/R-coin). Let's also add variables for Chain R, R_f, R_r, and R_d, which correspond to those we already have for Chain L. There's a symmetry between chains L and R, so we already know that ${}^{R_d}/_{R_r}$ gives us R-hashes/R-coin.

In theory, can an exchange rate help us convert between R-hashes/R-coin and L-hashes/L-coin?

[19]Attribution: I learned this, its significance for epistemology, and the importance of *goals* and *context* when evaluating *ideas*, from the philosopher Elliot Temple — particularly via his Critical Fallibilism Course (Elliot Temple; 2020) (which is difficult to cite directly). His article Multi-Factor Decision Making Math (Elliot Temple; 2021) details many related concepts and is a good starting point.

[20]Note that the terms *coin* and *root token* are synonymous. The choice of *coin* over *root token*, for these sections, is for practicality — we'll see this term *a lot*.

2.3.2 Converting Block-Weights

Can we find some function, $\mathrm{ConvWork}_{R\to L}(w)$, that converts R-hashes to L-hashes?

$$
\begin{array}{rlll}
 & \frac{L_d}{L_r} & \frac{\text{L-hashes}}{\text{L-coin}} & \\
\implies & \frac{L_d}{L_r}\cdot X_{R\to L} & \frac{\text{L-hashes}}{\text{R-coin}} & \text{Multiply by } X_{R\to L} \\
\implies & \frac{L_d}{L_r}\cdot X_{R\to L}\cdot \frac{R_r}{R_d} & \frac{\text{L-hashes}}{\text{R-hash}} & \text{Divide by } {}^{R_d}/_{R_r} \qquad (1) \\
\therefore \mathrm{ConvWork}_{R\to L}(w) = & \frac{L_d}{L_r}\cdot X_{R\to L}\cdot \frac{R_r}{R_d}\cdot w & \text{R-hashes} \to \text{L-hashes} & \qquad (2)
\end{array}
$$

With Equation 1, **we have just found our first constant of conversion for <u>block-weight</u>.**

Aside Equation 1 has a natural symmetry. It's worth noting for later.

What's going on here? We start out by observing ${}^{L_d}/_{L_r}$ gives us a value in units of $\frac{\text{L-hashes}}{\text{L-coin}}$. This is a constant of conversion from L-coins to L-hashes for a given moment — if some miner earned x L-coins today, then $x \cdot {}^{L_d}/_{L_r}$ would tell you roughly how many hashes were done to earn that reward. Next, we multiply by the exchange rate to find the constant of conversion for $\frac{\text{L-hashes}}{\text{R-coin}}$. Then, we divide by ${}^{R_d}/_{R_r}$ to find the constant of conversion for $\frac{\text{L-hashes}}{\text{R-hash}}$. If we multiply this constant of conversion by a value of R-hashes, then we'll end up with a value of L-hashes. It tells us *the relative weight* contributed to each network by each hash performed. Finally, we can deduce the function $\mathrm{ConvWork}_{R\to L}(w)$ which takes a value of R-hashes and returns a value of L-hashes.

Let's sanity check this.

Term **Root Token (RT)**: *aka* ***Coin***. The typically sole network-level token required by blockchain protocols. e.g., Bitcoin has BTC, Ethereum has ETH, Polkadot has DOT, Cardano has ADA, Amaroo has ROO, etc.

Consider two blockchains (L and R) that are *very* similar to Bitcoin. Unless otherwise specified, the chains are identical. Here are the key assumptions:

- Both L and R started on the same day, with the same block rewards (in their respective root tokens), block frequencies, and inflation schedules.
- L and R have equal money supplies, and the exchange rate has been stable at $X_{R\to L} = 3$ L-coins/R-coin.
- L and R use different PoW algorithms, L uses something like Scrypt (similar to Litecoin) and R uses something like SHA256 (similar to Bitcoin).
- ASIC/FPGA mining doesn't exist yet, but GPU mining does.
- (In this thought experiment) the best GPUs for mining Scrypt and SHA256 are of the same brand and model — i.e. the same supply is responsible for the hardware of *all* miners, regardless of which chain they mine.
- There's no comparative advantage between GPU makes/models — i.e., a miner can't increase their revenue by cleverly organizing which GPUs mine which networks.
- The cost of running both L and R nodes is negligible.
- L and R have perfect difficulty adjustment algorithms.
- The miner(s) used in this thought experiment are small relative to the total population of miners — their choices don't meaningfully impact network hash-rates or difficulty adjustments.

What should we expect regarding the conversion of work? To start with, let's note that GPU miners could work on either chain — good hardware for one chain is good hardware for the other, too. We know that L and R's block rewards (in root tokens) and block frequencies are the same

— so the exchange rate is going to play a dominant role in RoI (since the only other difference is difficulty and hash-rate). If a miner could break even by making 30 L-coins, then they could also break even by making 10 R-coins. They'd need to make 3× as many L-coins as R-coins — that's the exchange rate. If L and R used the same hashing algorithm, then we could compare difficulties to see if this makes sense — does that miner make 3× as many L-blocks as they would R-blocks?

In this case, though, the difficulties are set for *different hashing algorithms* — so how many hashes can GPUs do for each hash? Say a GPU can do 7 SHA256 hashes for each 1 Scrypt hash. A miner that can do h Scrypt hashes/day should be able to do $7h$ SHA256 hashes/day. That same miner should be able to make $h \cdot {}^{L_r}/_{L_d}$ coins per day — L's coins per block, divided by L's difficulty (hashes per block) gives us a constant of conversion with units L-coins/L-hash. Of course, the miner could, instead, mine on R, thus making $7h \cdot {}^{R_r}/_{R_d}$ coins per day. How do we know which is better? We use the exchange rate, of course!

If miners could swap from their current chain to the other chain and *increase their revenue*, then we should expect some to do that. In turn, we expect each chain's difficulty to change, reflecting that change in participation. If some miners (on the whole) moved from L to R, then we'd expect L's difficulty to decrease and R's difficulty to increase, corresponding to how many miners moved. Since this is an *arbitrage opportunity* (for miners), we expect that any profitability gap will quickly be closed. Thus, we can say that a miner's revenue is *equal* regardless of which chain they're mining: $\text{Revenue}_L = \text{Revenue}_R$ when measured in the same units.

$$\begin{aligned}
\text{Revenue}_L &= h \cdot \frac{L_r}{L_d} && \text{L-coins} && \text{Revenue on } L && (3)\\
\text{Revenue}_R &= 7h \cdot \frac{R_r}{R_d} \cdot X_{R\to L} && \text{L-coins} && \text{Revenue on } R \text{ in L-coins} && (4)\\
\text{Revenue}_L &= \text{Revenue}_R && \text{L-coins} && \text{The equality we set above} &&\\
\therefore h \cdot \frac{L_r}{L_d} &= 7h \cdot \frac{R_r}{R_d} \cdot X_{R\to L} && \text{L-coins} && \text{Equation 3 and Equation 4} &&\\
\frac{L_r}{L_d} &= 7 \cdot \frac{R_r}{R_d} \cdot X_{R\to L} && \frac{\text{L-coins}}{\text{L-hash}} && \text{Divide by } h &&\\
\frac{R_d}{L_d} &= 7 \cdot \frac{R_r}{L_r} \cdot X_{R\to L} && \frac{\text{R-hashes} \cdot \text{L-blocks}}{\text{L-hash} \cdot \text{R-block}} && \text{Multiply by } \frac{R_d}{L_r} &&\\
\frac{R_d}{L_d} &= (7 \cdot 3) && \frac{\text{R-hashes} \cdot \text{L-blocks}}{\text{L-hash} \cdot \text{R-block}} && \text{Sub } X_{R\to L},\ R_r, \text{ and } L_r && (5)\\
R_d &= 21 L_d && \frac{\text{R-hashes}}{\text{R-block}} && \text{Multiply by } L_d && (6)
\end{aligned}$$

So, the ratio of *difficulties* should be $21 \frac{\text{R-hashes}\cdot\text{L-blocks}}{\text{L-hash}\cdot\text{R-block}}$; or, R's difficulty *value* should be 21× L's difficulty *value*.

Why did the substitution of $X_{R\to L}$, R_r, and L_r (Equation 5) equal 3, though? First, notice that the units did not change with that operation. Next, we know the exchange rate $X_{R\to L} = 3$; we said so earlier. So it must be that ${}^{R_r}/_{L_r} = 1$. This simplifying step is only possible because we began *calculating* numerical values. We said earlier that L and R have — numerically — identical block rewards, so it must be that ${}^{R_r}/_{L_r} = 1$ *in this case.*

Let's consider Equation 2 in light of the above.

$$\begin{aligned}
\text{ConvWork}_{R\to L}(w) &= \frac{L_d}{L_r} \cdot X_{R\to L} \cdot \frac{R_r}{R_d} \cdot w && \text{R-hashes} \to \text{L-hashes} && \text{Equation 2}\\
&= \frac{1}{21} \cdot \frac{R_r}{L_r} \cdot X_{R\to L} \cdot w && \text{L-hashes} && \text{Sub } {}^{L_d}/_{R_d} \text{ from Equation 5}
\end{aligned}$$

$$= \frac{3}{21} \cdot w = \frac{w}{7} \qquad \text{L-hashes} \qquad \text{Sub } \frac{R_r}{L_r} \cdot X_{R \to L} \text{ as before}$$

We said this was true earlier — 1 L-hash is worth 7 R-hashes. Thus far, we have not yet found an inconsistency (i.e., we don't yet have a reason to think this won't work).

There is, however, an inconsistency lurking. Consider:

$$\frac{R_r}{L_r} \cdot X_{R \to L} \qquad \frac{\text{L-blocks}}{\text{R-block}}$$

$$\frac{L_f}{R_f} \qquad \frac{\text{L-blocks}}{\text{R-block}}$$

$$\text{But!} \qquad \frac{L_f}{R_f} \neq \frac{R_r}{L_r} \cdot X_{R \to L} \qquad \frac{\text{L-blocks}}{\text{R-block}} \tag{7}$$

These two values are **not** equal (or comparable), and nothing we've said implies that they should be! There are *qualitative differences* between the two that is not represented in the current units. On the one hand, we have something like *relative block frequencies,* and on the other we have something like *a ratio of the weight or value of block creation.* But they have the same units! What's going on? How do we know whether a constant of conversion *works* for our purposes?

2.3.3 Hold Up! We Need to Talk About $^{L_f}/_{R_f}$ and $^{R_r}/_{L_r} \cdot X_{R \to L}$

Aside This section regards some subtle ideas about when conversions work (i.e., give meaningful results), and when conversions don't. It's worth spending some time on these ideas because *when and how* you can convert is not always obvious. But, we *must* understand this to construct a meaningful method of converting block-weight — which PoR *requires.*

Let's consider some units with *real-world* interpretations. What can L-blocks/R-block *mean?*

- *Relative block frequencies* or *relative confirmation rates* — This has real-world meaning: Eth1 produces approximately 40 Ethereum-blocks in the same period (measured in seconds) that Bitcoin produces 1 Bitcoin-block.
- *Relative block weights* — This has real-world meaning: how much harder is it to generate a block on one network vs another network?
- *Relative confirmations* — This has real-world meaning: how many confirmations does one network take, compared to another, to reach equivalent security?[21]

Intuitively, *relative block weights* and *relative confirmations* sound related. If blocks on L are 5× heavier than blocks on R, then we'd have a constant of conversion of $^1/_5$ L-blocks/R-block; and a chain of 5 R-blocks would be *roughly* as hard to create as a chain of 1 L-block. So $^1/_5$ seems like a reasonable estimate for *relative confirmations*, too.[22]

Naively, *relative block frequencies* seems to be in the same units as the other two: L-blocks/R-blocks; but they *cannot* be in the same units as *the values mean different things.* Let's consider *relative confirmation* ***rates*** particularly. What happens if we assume that *seconds* on each chain aren't the

[21]"Equivalent security" means that a doublespend attempt on one network is just as risky, costly, etc, as a doublespend attempt on the other network. To do this comparison, we start by picking some q for the attacker on L, a transaction value (in L-coins), L's block reward, and then find the boundary of attack-viability (measured in L-confirmations). The boundary of attack-viability is where rules of thumb around confirmation times come from, e.g., *for Bitcoin, a transaction is safe after 6 confirmations.* Next, we consider an *equivalently valuable* transaction on R (converting via the exchange rate), and an equivalent attacker (using Equation 2 to convert). How many confirmations are needed on R so that $\mathrm{P}_L(\text{attack success}) = \mathrm{P}_R(\text{attack success})$?

[22]Due to the dynamics of confirmations, we can't directly compare chain-segments like this, *generally* speaking — this example is here to help give you an intuition. The reason we can't directly compare in this way is that simply *having more confirmations* is worth something in and of itself. The relationship is not linear. See Analysis of hashrate-based double-spending (Meni Rosenfeld; 2012) for more.

same thing, i.e., the units of *confirmation rate* are L-blocks/L-second (or R-blocks/R-second)?[23] Crucially, we can *not* cancel the *seconds* anymore:

$$? = \frac{L_f}{R_f} \qquad \frac{\text{L-blocks} \cdot \text{R-seconds}}{\text{L-second} \cdot \text{R-block}} \qquad \text{Relative confirmation rates} \tag{8}$$

$$? = \frac{R_r}{L_r} \cdot X_{R \to L} \qquad \frac{\text{L-blocks}}{\text{R-block}} \cdot \frac{\text{R-coins} \cdot \cancelto{1}{\text{L-coins}}}{\cancel{\text{L-coins}} \cdot \text{R-coins}} \qquad \text{Relative block weights via } X_{R \to L} \tag{9}$$

We can see that Equation 8 and Equation 9 are now obviously not comparable.

The reason that L_f/R_f did not make sense before is that we *were not including all necessary context!* There is *implicit context* in some properties of blockchains — *participation*. Values like L_f/R_f — when used to measure the *target block frequency* — *do not factor in participation*; the target block time is usually a *constant*, so it can hold no *network-specific context*.

Where does this network-specific context come from? How is it separated from "world" context — like target block frequencies? How is the network-specific context maintained over time? The answer to all three questions is the same: the **Difficulty Adjustment Algorithm** (DAA).

Term

Difficulty Adjustment Algorithm (DAA): An algorithm which updates its chain's difficulty as valid blocks are produced. The *output* of a DAA is *context laden* — units take on *additional context*.

DAA's typically work like this: calculate a *ratio* by which to adjust (multiply) the prior difficulty, based on a *target* block production rate and the *measured* block production rate.

Bitcoin, for example, adjusts its difficulty every 2016 blocks.[24] A ratio is found by multiplying the previous difficulty (D_{prev}) by the target duration (Δt_{target}) for 2016 blocks and dividing by the actual duration (Δt_{actual}) of the last 2016 blocks.[25] Note that the units of Δt_{actual} are B-seconds/(2016 B-blocks), and the units of Δt_{target} are seconds/(2016 blocks).

DAA's are special: they are the means by which *context* is added. DAA's don't explicitly deal with this context though — it's not mentioned in the algorithm itself. The key to a DAA's success is that it operates *relative to a past state that is already context laden.* So DAA's don't need to have any special awareness of context, just that multiplying the past difficulty by a *particular ratio* will adjust the *confirmation rate* to align with the *target block frequency.* It's an *incremental and ongoing process.* Since DAA's don't have initial conditions, there's no bootstrapping concern. To function, a DAA only needs to say how the difficulty should *change*; it doesn't need to know what it actually *is.* We will use the subscript $W \to B$ to denote the idea of converting between some *world* context, and the *network context* (of Bitcoin).

$$\text{Note that:} \quad \text{ConversionConst}_{W \to B} = \frac{\Delta t_{\text{target}}}{\Delta t_{\text{actual}}} \qquad \frac{\text{B-blocks} \cdot \text{seconds}}{\text{B-second} \cdot \text{block}}$$

$$\text{Bitcoin's DAA:} \quad \text{NextWork}_{W \to B}(D_{\text{prev}}) = \frac{\Delta t_{\text{target}}}{\Delta t_{\text{actual}}} \cdot D_{\text{prev}} \qquad \frac{\text{B-hashes} \cdot \text{seconds}}{\text{B-second} \cdot \text{block}} \cdot \frac{\cancelto{1}{\text{B-blocks}}}{\cancel{\text{B-block}}}$$

When a DAA adds context, it converts blocks ↔ B-blocks, and seconds ↔ B-seconds. Alternatively, it could *strip* context. Either way works because the DAA acts as a boundary of the convertible context in both cases. This means that *when converting* we can cancel B-seconds with seconds, B-blocks with blocks, etc.

$$\text{Alt. with context:} \qquad \text{NextWork}_{W \to B}(D_{\text{prev}}) = \frac{\Delta t_{\text{target}}}{\Delta t_{\text{actual}}} \cdot D_{\text{prev}} \quad \frac{\text{B-hashes}}{\text{B-block}} \tag{10}$$

[23] Alternatively, you could assume that confirmation rates are *always* in the same units (i.e., *generic* blocks/second). That will yield similar results; the logic basically works either way with some minor tweaks. The important point is that the units of L_f/R_f are **not** L-blocks/R-block.

[24] Note: in Bitcoin, a difficulty of 1 corresponds to 2^{32} hashes.

[25] Note: in practice the ratio is clamped between 1/4 and 4. See Bitcoin's `src/pow.cpp` for the implementation.

$$\text{Divide by } D_{\text{prev}} \quad \Longrightarrow \quad \text{ConversionConst}_{W\to B} = \frac{\Delta t_{\text{target}}}{\Delta t_{\text{actual}}} \qquad \frac{\text{B-hashes} \cdot \cancelto{1}{\text{B-blocks}}}{\cancel{\text{B-hash}} \cdot \text{B-block}}$$

Term

Convertible Context: The boundary of a group of values that are mutually convertible. Within a convertible context, all values must be of the same *scale* or have known exact scaling factors.

The general case of a DAA's relationships (flows of *information* and *context*) are diagrammed in Figure 9.

How do we know that *both* blocks and seconds become context laden via a DAA, though? Let's consider what $^{L_f}/_{R_f}$ means for the possible combinations of context laden values and note whether the meaning works for conversion or not (i.e., whether using it *appropriately* as a constant of conversion, or scaling factor, will produce sensible results).

$$\text{No context:} \quad \frac{L_f}{R_f} \qquad \text{(unitless)} \qquad \text{works} \tag{11}$$

$$\text{Context laden blocks:} \quad \frac{L_f}{R_f} \qquad \frac{\text{L-blocks}}{\text{R-block}} \qquad \text{fails} \tag{12}$$

$$\text{Context laden seconds:} \quad \frac{L_f}{R_f} \qquad \frac{\text{R-seconds}}{\text{L-second}} \qquad \text{fails} \tag{13}$$

$$\text{Both context laden:} \quad \frac{L_f}{R_f} \qquad \frac{\text{L-blocks} \cdot \text{R-seconds}}{\text{L-second} \cdot \text{R-block}} \qquad ? \tag{14}$$

We've seen Equation 11 and Equation 12 before. The first represents the ratio of block frequencies (unitless) — that's straightforward and works. The second has units L-blocks/R-block, which sounds like it should be the ratio of block *weights* — but it's clear that it *isn't* that. (So this conversion method fails.)

Equation 13 has weird units, though. R-seconds/L-second means something like: the relative participation of each network compared with a recent past state; i.e., the ratio of the ratios of each network's *actual* block production compared to its *target* block production. (This conversion method also fails.)

Equation 14 measures something like *relative weighted confirmation rates*. It's not clear if is useful or not, but we *do* know that *no other* value we have access to has context laden seconds as a unit. How can we use it to convert between anything meaningful if context laden seconds can't be canceled via some conversion? (Do we even *need* to ever use those units, anyway?)

In general, it seems like the safe option is *not to use L_f or R_f when converting work* — unless we have some *specific, context-driven* explanation for why it's okay in that case.

How do these ideas of context laden values work when converting values *between* these network-contexts? This is diagrammed in Figure 10.

In essence, an exchange rate provides meaningful conversion between L-coins and R-coins. Converting in this way *does not drop context*. Since network context is respected, we can use an exchange rate to build a meaningful constant of conversion *across networks*.

Aside

With regard to DAAs, it should be noted that Bitcoin's was the first, and the method has some undesirable properties. I quite like the algorithm named DAA-2 (which is used by Bitcoin Cash) in An Economic Analysis of Difficulty Adjustment Algorithms in Proof-of-Work Blockchain Systems (Noda, Okumura, Hashimoto; 2020). Experimentally, it seems to work well with Section 4.8.

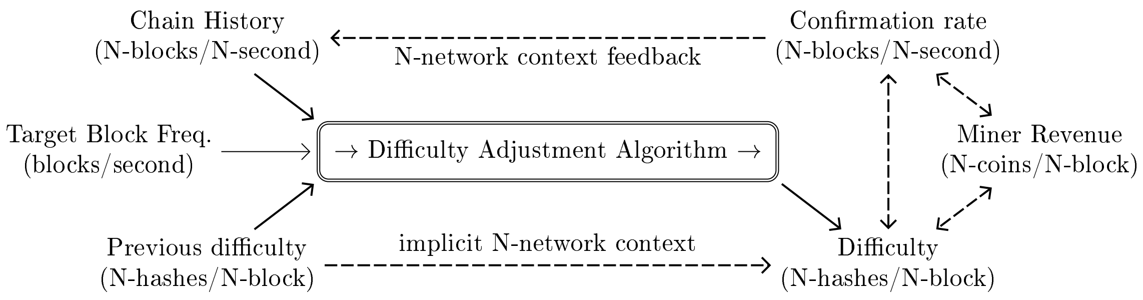


Figure 9: The difficulty adjustment algorithm governs the relationship between the inputs: the previous difficulty, the target block frequency, and network participation (chain history); and the output: the network difficulty. The DAA is how *confirmations* and *coins* become **laden** with *implicit context.* If we don't account for this *implicit context* then our conversions will be nonsensical. The implicit context is *network participation* — thus, `N-` prefixes the units which are *context laden.* Thick arrows indicate *network context*, and thin arrows indicate *world context.* Solid arrows show the *flow* of *information.* Dashed arrows show the *flow* of *context.* Two-way arrows ($\longleftrightarrow$) link two values that are *convertible.* The collection of values mutually linked by two-way arrows define the *convertible context.* Values can only be converted when there is a direct two-way path between them.

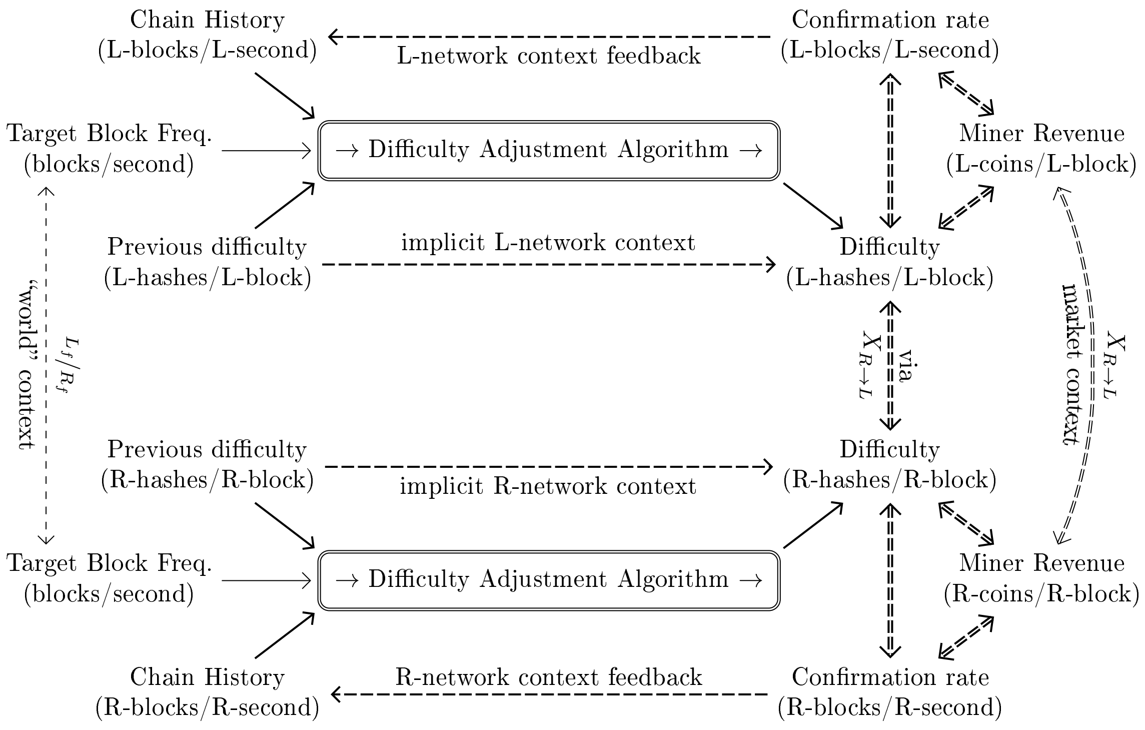


Figure 10: How are the convertible contexts of two different networks related? Without the market context, there's no conversion path that allows for the conversion of work — the conversion path between difficulties is a *consequence* of $X_{R\to L}$ (the exchange rate). This is the same convertible context that miners use to determine which network is most profitable for them. Double-lined arrows indicate *market context.* Thin single-lined dashed arrows indicate *world context.* Notice that the convertible properties which we are interested in (such as L_d and R_d) use *thick, double-lined, and dashed* two-way arrows, indicating that we are using network context *and* market context to convert block-weight.

2.3.4 Conversions and Sums

We know that, after conversion, we can sum work from two different chains. Are there any *other* values (in units other than L-hashes) that we can sum up, though? When we're *summing* weights as part of calculating chain-weight (e.g., that of Algorithm 2, or Algorithm 6), do we need to sum *L-hashes*? Well, no. We only need to *end up* with L-hashes.

Consider the case for a two-stage linear conversion method. That is: we convert the input into some common units (which could be anything), then we convert those common units into the final units. If both partial-conversions are *linear*, then we must have a situation like this:

$$\begin{aligned} \mathrm{Convert}_{L\to R}(\dots) &= \mathrm{Conv}_1(\mathrm{Conv}_2(\dots)) \\ &= V_1 \cdot \mathrm{Conv}_2(\dots) && \text{For some constant of conversion, } V_1 \end{aligned}$$

Let's sum multiple conversions, e.g., as done in Algorithm 2:

$$\begin{aligned} \sum_{i=0}^{n} \mathrm{Convert}_{L\to R}(\dots) &= \sum_{i=0}^{n} V_1 \cdot \mathrm{Conv}_2(\dots) \\ &= V_1 \cdot \sum_{i=0}^{n} \mathrm{Conv}_2(\dots) && \text{Factorize out } V_1 \end{aligned}$$

Thus, *any* common units, which are linearly convertible both from a reflecting chain's block and to local chain-work, can be used during summation.

Aside

Before we move on, let's consider:

$$\begin{aligned} &\frac{L_d}{L_r} \cdot X_{R\to L} && \frac{\text{L-hashes}}{\text{R-coin}} \\ &\frac{R_r}{R_d} \cdot X_{R\to L} && \frac{\text{L-coins}}{\text{R-hash}} && \text{Similarly} \\ \therefore \mathrm{ConvReward}_{R\to L}(w) = &\frac{R_r}{R_d} \cdot X_{R\to L} \cdot w && \text{R-hashes} \to \text{L-coins} && (15) \end{aligned}$$

Is it possible that we can convert chain-work *via summing block rewards?*

2.4 Conversion Contexts

What blockchain contexts can facilitate the conversion of block-weight?

Whatever contexts we find, we will need to figure out a way to get the exchange rate that is *at least as secure* as the consensus algorithms (otherwise we'd be introducing a new weakest-link). That can't be too hard, right?

Can we *avoid* that exchange rate, though? Well, there is a context where $X_{R\to L} = 1$: **when L-coins** $\equiv$ **R-coins**, i.e., both chains use the same root token. In that case, $^{L_d}/_{L_r} \cdot {}^{R_r}/_{R_d}$ gives us L-hashes/R-hash directly.

2.4.1 A Single Root Token Across Multiple Chains

The first example context is *one network split over two chains.* Both chains use the same *root token.* There are, naturally, questions to answer, like *How are block rewards managed?* and *How can the root token be on two blockchains at once?*

Let's consider the following *restricted case* first: the chains (L and R) have something like a two-way peg between them, ensuring that no coins are ever incorrectly created or destroyed.

In this case, for a single R-block, Equation 2 collapses to:

$$
\begin{aligned}
\text{ConvWork}_{R\to L}(w) &= \frac{L_d}{L_r} \cdot \frac{R_r}{R_d} \cdot w && && \text{Since } X_{R\to L} = 1 \\
\implies \text{ConvWork}_{R\to L}(R_d) &= \frac{L_d}{L_r} \cdot R_r && \text{L-hashes} && w = (R_d \cdot 1) \text{ R-hashes} \\
\implies \text{ConvWork}_{R\to L}(R_d) \cdot \frac{L_r}{L_d} &= R_r && \text{L-coins} && \begin{array}{l}\times \text{ L-coins/L-hashes} \\ \text{Weight of an R-block in L-coins}\end{array}
\end{aligned} \tag{16}
$$

So, the weight of an R-block measured in L-coins is exactly equal to the R chain's block reward. This, of course, makes sense. L-coins are fungible with R-coins in this context, so, naturally, the weight-in-coins of an R block is its own block reward.

If we convert like this (treating 1 R-block as worth $R_r \cdot 1$ L-coins) then we'll we need to convert the result to L-hashes *eventually*, but, if we have multiple reflections to process, we can calculate interim values (with units of L-coins) and sum them before that final conversion. It's worth remembering that block-weight is measured *per block*. Since we're only converting with respect to *a single block* at any one time, the *per block* part of the resulting units *cancels out* — e.g., Equation 16.

Let's use this context to write our first WeightOf function: Algorithm 3 (we'll still return work in L-hashes, though, not L-coins).

Aside Note: the algorithms we define here will be general — they'll work for *both* the local and any reflecting chains.

Algorithm 3 A WeightOf for networks of a single root token.

```
procedure WeightOf($B_i$, state)                       ▷ The weight of a reflecting block in L-hashes
    $t \leftarrow$ BlockTimestamp($B_i$)
    $R \leftarrow$ ChainOf($B_i$, state)                    ▷ Either the local or a reflecting chain
    $R_r \leftarrow$ BlockRewardOfAt($R$, $t$, state)          ▷ L-coins/R-block — by definition
    $L_d \leftarrow$ LocalDifficultyAt($t$, state)
    $L_r \leftarrow$ LocalBlockRewardAt($t$, state)
    return $R_r \cdot (L_d \div L_r)$                     ▷ See Equation 16 (Note: $R_d$ cancels out)
end procedure
```

But where do we get the reflecting chain's block reward? To start with, we can't let each chain *set* its own reward, that would break the two-way peg. It also means *the reflecting chain shouldn't be the source of that data.* We could retrieve it from state (if it's already calculated), but for the sake of this example, let's calculate it in the WeightOf function.

Right now, we don't have enough *context* to know what to do; *what contexts would be useful?*

What *purpose* does a block reward fulfil? It must be something about *security*, because that's *why* miners are given an incentive to mine (the *how* is via the *block reward*). If we have multiple chains (with the same root token), why should one chain be more secure than another? One reason is because *more commerce happens there;* i.e., more of the network relies on *that* chain compared to the other one.

What would make sense, given the context of a commerce-imbalance, to base the block reward on? A straightforward answer (which can be globally known, too) is *the ratio of root tokens on each chain.* We can have a *network wide* rule that there is some *network wide inflation rate*, and each chain's block reward is proportional to the *ratio* of coins on that chain relative to the entire supply. This way, the global inflation rate is predictable, even though the number of coins produced *on each chain* (via block rewards) is variable. Moreover, with this method we should expect that the *value to the community* of each chain roughly matches the *distribution of the community's activities*, and the *distribution of security*, too. This method is also consistent with the idea of *equal work for* *equal reward.* We can rely on that idea because *equality of work* is a result of the market (for block rewards) formed by participating miners.

If the network uses this rule to determine block reward, then, for either chain C, what can we say? Let: I (coins/s) be the network-wide inflation rate; G_t (coins) be the network-wide root token supply; and C_t (coins) be the number of root tokens on chain C. C_f (blocks/s) is the block production frequency of chain C.

$$C_r \cdot C_f = \frac{C_t}{G_t} \cdot I \qquad \frac{\text{C-coins}}{\text{second}} \qquad \text{As set above}$$

$$\implies C_r = \frac{C_t \cdot I}{G_t \cdot C_f} \qquad \frac{\text{C-coins}}{\text{C-block}} \tag{17}$$

$$\therefore \frac{R_r}{L_r} = \frac{R_t \cdot I}{G_t \cdot R_f} \cdot \frac{G_t \cdot L_f}{L_t \cdot I} \qquad \frac{\text{L-blocks}}{\text{R-block}} \qquad \text{Via Equation 17}$$

$$\frac{R_r}{L_r} = \frac{R_t \cdot L_f}{L_t \cdot R_f} \cdot \frac{G_t \cdot \cancelto{1}{I}}{\cancel{G_t} \cdot I} \qquad \frac{\text{L-blocks}}{\text{R-block}} \qquad \text{by } \textit{definition} \tag{18}$$

Hold up! Didn't we just go over — in great detail — why we can't use block frequencies (L_f and R_f) in conversions? Why is it okay to use them now?

To answer this, let's consider *how* block rewards could be set network-wide so that they're consistent across all chains. Unlike a chain's difficulty, the block reward needs to be consistent with some global inflation rate. Equation 18 implies that $R_r \cdot R_f \cdot L_t = L_r \cdot L_f \cdot R_t$, so in some cases *we don't need to know the global state, just that the relationship is enforced.* That said, if I is known across the network (i.e., it is constant or a well known function) then calculating G_t is trivial: $G_t = I \cdot l + G_{t_0}$, where l is the lifetime of the network (in seconds), and G_{t_0} is any initial root token supply that was not created via inflation (e.g., via some token generation event).

Could we maintain consistency across chains via something like a **Reward Adjustment Algorithm** (RAA)? If each chain commits to the input values (e.g., via the inclusion of C_t in their block-headers), then an RAA could be evaluated using only the header-chain (similar to the DAA). This way, if a miner (or a chain) tries to subvert or attack the RAA, it's trivially detectable by reflecting chains.

Term

Reward Adjustment Algorithm (RAA): An algorithm which updates the block reward of each chain in a *network* of chains that share a root token. Similar to a DAA, the *output* of an RAA is *context laden.*

If we use an RAA, then the RAA *output* could become *context laden* (depending on the RAA's input parameters). In the case of a global inflation rate, *this means that chains would rely on a common meaning of "seconds"!* So it is by *construction and definition* that the block frequency introduced in Equation 17 becomes part of the convertible context. Thus, we can *now* include L_f and R_f, though we couldn't before.

Note that we also *prove* Equation 7 for this case. We know that — when L_f and R_f are in the convertible context — we can use them via a *scaling factor* that typically $\neq 1$. For example: to convert $^{L_f}/_{R_f}$ to units of L-blocks/R-block, it must be scaled by $^{R_t}/_{L_t}$ (which is contextual and unitless). We can find a scaling factor *here* (but not generally) because we *included it in a protocol-level relationship* (Equation 17).

This new situation and context are diagrammed in Figure 11.

Let's use Equation 18 to create our next WeightOf algorithm for this context of a *single root token* (SRT) — Algorithm 4.

$$\frac{R_t \cdot L_f}{L_t \cdot R_f} \cdot L_d \qquad \frac{\text{L-hashes}}{\text{R-block}} \qquad \text{Via Equation 18}$$

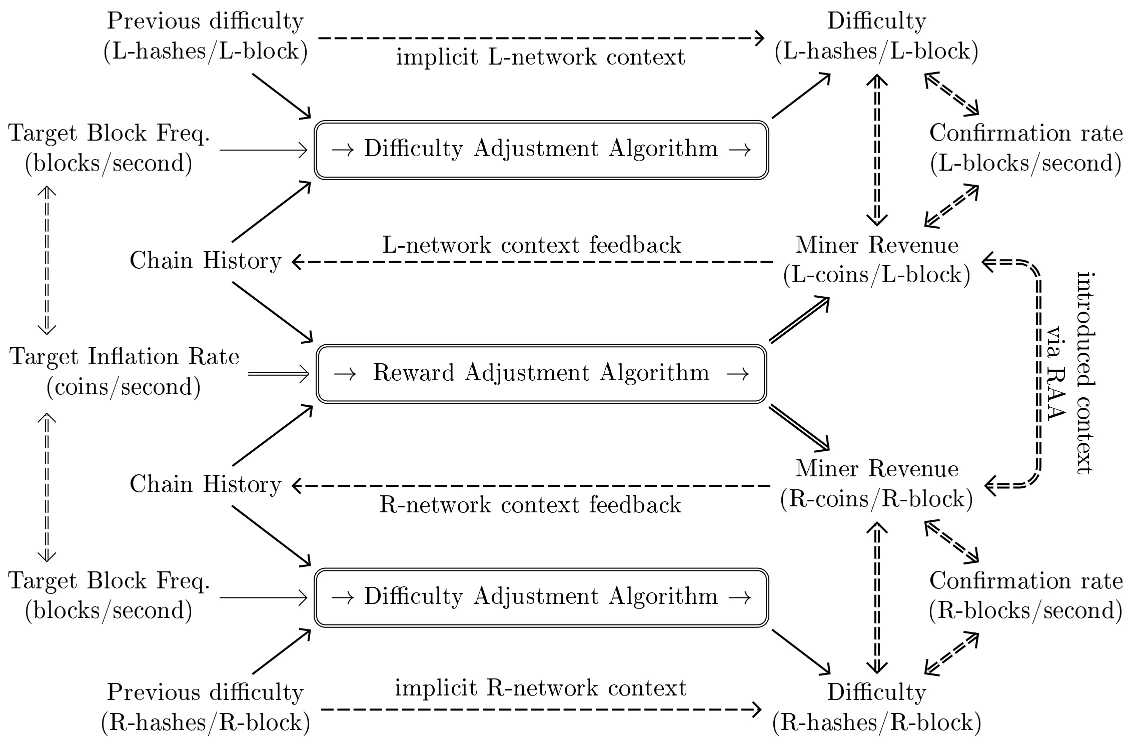

Previous difficulty (L-hashes/L-block)
implicit L-network context
Difficulty (L-hashes/L-block)
Target Block Freq. (blocks/second)
→ Difficulty Adjustment Algorithm →
Confirmation rate (L-blocks/second)
Chain History
L-network context feedback
Miner Revenue (L-coins/L-block)
introduced context via RAA
Target Inflation Rate (coins/second)
→ Reward Adjustment Algorithm →
Chain History
R-network context feedback
Miner Revenue (R-coins/R-block)
Target Block Freq. (blocks/second)
→ Difficulty Adjustment Algorithm →
Confirmation rate (R-blocks/second)
Previous difficulty (R-hashes/R-block)
implicit R-network context
Difficulty (R-hashes/R-block)

Figure 11: How does implicit context change when considering networks of a single root token? In this case, because we set a network-wide inflation rate (Equation 17), by *definition* L_f and R_f have meaning in the convertible context. In essence: both L and R have convertible factors that are dependent on the same meaning of "seconds". The collection of values mutually linked by two-way arrows ($\longleftrightarrow$) define the *convertible context*. Double-lined arrows indicate the *new* context used in the RAA (which is related to the world context). Thick Double-lined arrows indicate the new hybrid context which allows for conversion (for multi-chain networks of a single root token). As before, dashed arrows indicate the flow of context.

Algorithm 4 A WEIGHTOF function for networks of a single root token based on the ratio of root tokens that it hosts.

```
procedure WEIGHTOF($R_i$, $state$)                          ▷ Returns L-hashes
    $t \leftarrow$ BLOCKTIMESTAMP($R_i$)
    $R \leftarrow$ CHAINOF($R_i$, $state$)
    $R_f \leftarrow$ BLOCKFREQUENCYOFCHAIN($R$, $state$)     ▷ Block production Hz of $R$
    $R_t \leftarrow$ ROOTTOKENSONCHAINAT($R$, $t$, $state$)  ▷ Number of root tokenson $R$ at $t$
    $L_d \leftarrow$ LOCALDIFFICULTYAT($t$, $state$)
    $L_f \leftarrow$ LOCALBLOCKFREQUENCY($t$, $state$)
    $L_t \leftarrow$ LOCALROOTTOKENS($t$, $state$)
    return CONVSRT$_{R\to L}$($L_d$, $L_f$, $L_t$, $R_f$, $R_t$)
end procedure

procedure CONVSRT$_{R\to L}$($L_d$, $L_f$, $L_t$, $R_f$, $R_t$)   ▷ Returns L-hashes for 1 R-block
    return $R_t \cdot L_f \cdot L_d \div L_t \div R_f$               ▷ See Equation 19
end procedure
```

$$\therefore \text{ConvSRT}_{R\to L}(b) = \frac{R_t \cdot L_f}{L_t \cdot R_f} \cdot L_d \cdot b \qquad \text{R-blocks} \rightarrow \text{L-hashes} \tag{19}$$

Aside These and the following conversions all collapse nicely if we set $R \equiv L$. This is due to the symmetry mentioned at the start of Section 2.3.2.

2.4.1.1 Degenerate Case

There is an even simpler case when the same hashing algorithm is used and when the rewards and difficulties are equal. In this case, the conversion to/from remote units is trivial, since these are now, *by definition,* identical to local units. Equivalently: all constants of conversion between corresponding values are 1. Subsequently, Equation 16 collapses to:

$$\text{ConvWork}_{R\to L}(w) = w \qquad \text{L-hashes} \qquad \text{If } R_d = L_d, R_r = L_r \tag{20}$$

While this works fine for PoR, it offers little value for this discussion on conversion theory.

2.4.2 Different Root Tokens with a DEX

Aside Before we explore this context, I should mention that we are discussing it primarily so that we can generalize our understanding of the conversion of work. This context presents substantial challenges that the SRT context does not.

Our starting point will be Equation 2:

$$\text{ConvWork}_{R\to L}(w) = \frac{L_d}{L_r} \cdot X_{R\to L} \cdot \frac{R_r}{R_d} \cdot w \qquad \text{R-hashes} \rightarrow \text{L-hashes}$$

It's easy to see that we can work with this — of course, the WeightOf function needs access to R's difficulty and block reward, and the exchange rate, too. Do we *need* to use R_d, though? Surely *any* conversion to L-hashes will work, right?

Let's see:

$$\frac{L_d}{L_r} \cdot X_{R\to L} \cdot R_r \qquad \frac{\text{L-hashes}}{\text{R-block}}$$

$$\therefore \text{ConvDEX}_{R\to L}(b) = \frac{L_d}{L_r} \cdot X_{R\to L} \cdot R_r \cdot b \qquad \text{R-blocks} \rightarrow \text{L-hashes} \tag{21}$$

Okay, so we *can* convert to L's block-weight (L-hashes) *without knowing R's difficulty*. It's good to know that we have *options* there — there's *excess capacity* in the values we can convert between. If one value is not available, then we might not even need it. (This is relevant for Section 2.5.)

To move forward, we need to use $X_{R\to L}$ in a new WeightOf function — *where does that value come from?*

The value can't be hard-coded, or provided by a third party — these would introduce vulnerabilities. We need an accurate exchange rate for the moment of conversion, too — one that is reactive enough to remain up-to-date, and precise enough to be useful. It also needs to be *provable*; in practice, it needs to be *on-chain* and available to both chains (and they should agree on the exchange rate). If that wasn't the case, how could one chain validate the weight of another chain's PoRs? Ideally, the exchange rate should *already* be recorded in *both* chains. (This way, full nodes of either chain do not need extra data to calculate/verify chain-weight.)

However, there is a major loop-hole in some of the above requirements: they only matter for *mutual PoR*. What's different if we consider *one-way PoR* instead?

In the case of one-way PoR, the reflecting chain doesn't need to know much about the reflected chain, and the heavy lifting (like the burden of knowing exchange rates) can be offloaded to *the reflected chain only*. Provided that the source of the exchange rate is a *well-built*, protocol-level DEX, are there any real issues?

Let's consider this limited case first: one-way PoR where the *reflected* chain hosts a DEX between its root token, and the RT of the *reflecting* chain. Using Equation 21, we can write Algorithm 5.

Algorithm 5 A WeightOf function for one-way PoR between chains with different root tokens.

```
procedure WeightOf($B_i$, $state$)                    ▷ Convert reflecting weight to local via a DEX
    $t \leftarrow$ BlockTimestamp($B_i$)
    $R \leftarrow$ ChainOf($B_i$, $state$)
    $R_r \leftarrow$ BlockRewardOfChainAt($R$, $t$, $state$)          ▷ Block reward of $R$ at $t$
    $L_d \leftarrow$ DifficultyOfLocalAt($t$, $state$)
    $L_r \leftarrow$ BlockRewardOfLocalAt($t$, $state$)
    $X_{R\to L} \leftarrow$ GetLocalDexRateFromAt($R$, $t$, $state$)
    return ConvDEX$_{R\to L}$($L_d$, $L_r$, $R_r$, $X_{R\to L}$)
end procedure

procedure ConvDEX$_{R\to L}$($L_d$, $L_r$, $R_r$, $X_{R\to L}$)          ▷ Returns L-hashes for 1 R-block
    return $(L_d \div L_r) \cdot X_{R\to L} \cdot R_r$                 ▷ See Equation 21
end procedure
```

This *seems* okay; of course, it *still depends on the DEX*. That means that an attack on the DEX might be a way to attack the consensus algorithm. Did we just introduce a vulnerability?

If an attacker *reduces* the exchange rate (L-coins/R-coin) somehow, then that will *decrease* the weight of reflected blocks. For example, an attacker might sell *a lot* of L-coins all at once (or cause that to happen through traditional market manipulation). If the exchange rate goes down, then the *rate of work* (i.e., how fast chain-weight is accumulated) of Chain L will *seem* to decrease.

However, this isn't *always* relevant: if Chain R is *much* more secure than Chain L, then this doesn't change much in practice. Say that Chain R contributes $\sim 100\times$ as much chain-weight as L does. For an attacker targeting Chain L, reducing the exchange rate *by half* means that the weight of reflecting blocks is halved. So the attacker would need to add $\sim 50\times$ the *current* hash-rate of L to compete against the reflections from R. If the attacker was mining in *public*, that would increase the local difficulty by $\sim 50\times$, which *increases* the weight of reflecting blocks by $\sim 50\times$, too (so an attacker mining in public is self-defeating). Of course, the attacker *could* still mine in private,[26] provided the exchange rate stays low. Even though the attacker gets a 50% discount via market manipulation, this is still a very difficult attack (and very noticeable). In effect, attacking L is at least as difficult as attacking R.

If an attacker *increases* the exchange rate (L-coins/R-coin) somehow, then that will *increase* the weight of reflecting blocks. This does not help the attacker.

It seems that *in principle* there are some cases where it's safe to use ConvDEX$_{R\to L}$ for one-way PoR. We'll revisit this later in Section 3.4.1.

Aside

What about mutual PoR, though? Notice that Algorithm 5 is trivially generalized for mutual reflection by replacing GetLocalDexRateFromAt with a suitable ExchangeRateToLocalOfAt, i.e., a DEX that satisfies the requirements we specified earlier. So it seems like our *conversion method* should work; the weak-link is the DEX, not the conversion.
In the context of *Ultra Terminum* and *Amaroo*, this isn't an important problem to solve. After all, this is a paper about *consensus*, not trustless, decentralized, cross-chain, protocol-level markets. If *mutual PoR* is ever used to secure multiple chains with heterogenous tokens, this

[26] For the purpose of UT and Amaroo, this turns out to be a non-issue. See Section 3.2, Section 3.4.1, Section 4.8, and Section 4.9.

problem will need to be answered, though.

2.4.3 What About SPV?

Both contexts (SRT and DEX) require that participating chains can do on-chain SPV against one another. Chains need some ability to *introspect* reflecting chains — e.g., SRT requires that users can move root tokens between chains, and the DEX context requires two chains to agree on the exchange rate between their root tokens. Even without this requirement, some method of cross-chain communication is clearly desirable.

Eventually, we'll need to construct a method for SPV between mutually reflecting chains that works and is safe. However, there are still other problems that we have not yet solved, and the solutions may motivate certain blockchain designs over others. The difference between these designs will likely impact whether (and how) SPV can be done safely. So, attempting to solve the SPV problem at this point is premature.

We will proceed on the *assumption* that SPV is possible and easy to do in a reasonable time period, and we'll investigate the problem of SPV between reflecting chains in detail in Section 4.11.

2.5 Converting Confirmations

So far, we've considered PoW chains only. Conversion of chain-weight between PoW chains can work *if and only if* we can convert between *work* (i.e., hashes) done on each chain — given an appropriate context. For a given PoW block, the network knows exactly how much work is implied by that block — the expected number of hashes to produce it. Thus, for PoW chains, there is an exact conversion between *work and confirmations* (for some context at some point in time). Over short time-scales, this conversion ratio is approximately constant (in general it's a function that takes a timestamp as an input parameter). Thus, *chain-weight* (as represented in figures via Σ_w, e.g. Figure 23) can be represented either in something like *hashes* or *difficulty* **or** chain-weight can simply be in terms of *confirmations.*

Aside

If we convert *work to confirmations*, will we end up with something *incompatible and contradictory* to the traditional notion of "a confirmation"? There are definitely differences. For example: if we convert confirmations, then *we'll have non-integer confirmations*, and what does 0.88 confirmations mean? Is that less good than a normal confirmation?
This problem arises because *we're not actually converting work to confirmations*, per se: we're converting *another chain's work* into *equivalent-confirmations* relative to something. Equivalent-confirmations are another chains confirmations that have been *converted to be in terms of the local chain's confirmations.* Most likely, those equivalent-confirmations will be relative either to some known historical confirmation, or to that of the *current* block.

Why think about chain-weight in terms of *equivalent-confirmations* instead of *work*? There are a few reasons. First, *confirmations are general!* If we reason in terms of *confirmations* instead of *work*, then *maybe* we can apply these ideas to *other chains* that don't use PoW. Second, it *simplifies thinking.* The purpose of converting chain-weight is clearer and easier to reason about. Finally, it makes explicit the requirement that *we can only compare to a grounded context.*

There is no way to say X *work on* L *is worth* Y *work on* R without adding necessary context like *when* that conversion is happening. Confirmations (like work) require that grounding, since they need to be scaled when converting between different chains. What about confirmations from the same chain? Unlike work (which can be summed directly), confirmations always require conversion to a *known standard* — even when they're *from the same chain.* For example, we can say that the single confirmation provided by Bitcoin block 704610 is *equivalent* to approximately 19,893,045,000,000 genesis-confirmations.[27] The conversion-ratio is equal to the difficulty of block 704610. That is, it would take a chain of $\sim$ 20 trillion blocks, each with 1 genesis-confirmation

[27] A genesis-confirmation is relative to the Bitcoin genesis block — which had a difficulty of exactly 1.

worth of work, to match the weight of block 704610. Curiously, while this works for the *fork rule* (since it compares chain weights), it does not work when waiting for transaction confirmations — in that case, 20 trillion confirmations at 1 difficulty is *much* more secure than 1 confirmation at 20 trillion difficulty.[28]

Now, **converting confirmations,** how do we actually do it? If we want to convert confirmations, then we'll need to abstract away from the idea of *difficulty* in our conversion method.

$$\text{Consider:} \quad \frac{R_r}{L_r} \cdot X_{R\to L} \qquad \frac{\text{L-blocks}}{\text{R-block}}$$

$$\therefore \text{ConvBlocks}_{R\to L}(b) = \frac{R_r}{L_r} \cdot X_{R\to L} \cdot b \qquad \text{R-blocks} \to \text{L-blocks} \tag{22}$$

So $1\times$ R confirmations is worth $\left(\frac{R_r}{L_r} \cdot X_{R\to L}\right)$ L confirmations. Nice and simple.

2.5.1 Coins per Confirmation

Given a multi-chain network, could we measure block-weight in coins? It seems promising and elegant if it works, but does it have any real-world meaning?

One example where measuring chain-weight in coins does have some meaning is Section 2.4.1 (the SRT context). Let's consider this, starting with the conversion used in Equation 18.

$$C_r = L_r \cdot \frac{C_t}{L_t} \cdot \frac{L_f}{C_f} \qquad \frac{\text{L-coins}}{\text{C-block}} \qquad \text{Via Equation 18}$$

$$C_r \cdot \frac{C_f}{L_f} = L_r \cdot \frac{C_t}{L_t} \qquad \frac{\text{L-coins}}{\text{L-block}}$$

$$\therefore \sum_{C\in\{L,R\}} L_r \cdot \frac{C_t}{L_t} = L_r \cdot \frac{L_t + R_t}{L_t} \qquad \frac{\text{L-coins}}{\text{L-block}} \qquad \text{Sum coins (as a proxy for weight)} \tag{23}$$

$$L_r \cdot \frac{C_t}{L_t} = \frac{L_t \cdot I}{G_t \cdot L_f} \cdot \frac{C_t}{L_t} \qquad \frac{\text{L-coins}}{\text{L-block}} \qquad \text{Via Equation 17}$$

$$= \frac{C_t \cdot I}{G_t \cdot L_f} \qquad \frac{\text{L-coins}}{\text{L-block}}$$

$$\therefore \sum_{C\in\{L,R\}} \frac{C_t \cdot I}{G_t \cdot L_f} = \frac{L_t + R_t}{G_t} \cdot \frac{I}{L_f} \qquad \frac{\text{L-coins}}{\text{L-block}} \qquad \text{Sum coins (as a proxy for weight)} \tag{24}$$

What does Equation 23 imply if L and R are the only two chains in a context like Section 2.4.1? Notice that, in this case, $L_t + R_t = G_t$, the network-wide currency supply. One implication is that weight (measured in coins) effectively counts *how much of the full network* is contributing to Chain L's security — represented via the coins that were minted in those contributing blocks. It's easier to see in Equation 24 as the sum collapses to ${}^{I}/{}_{L_f}$.

If the all chains in the network are functioning well, we should expect that summing a chain's weight in coins *over the full history of the chain* should be close to the sum of all coins minted through block rewards. Of course, this is only useful over *multiple* chains. **If a single, traditional blockchain tried to do this, then all chain-weights would be basically identical!**[29] This

[28] For an explanation of *why*, see Analysis of hashrate-based double-spending (Meni Rosenfeld; 2012) and Section 4.7.

[29] This may be a new criticism of PoS. In essence: a blockchain needs something like a DAA to factor-in participation. PoS chains use *coins* instead of *hashes*, but *coins* will never provide a way to determine which chain has higher participation. Moreover, *coins* are actually a very *bad* way to measure participation (for a standalone PoS chain), because the *most valuable future network* is one where coins are being used for *actual trade*, and this must happen at the expense of the number of coins used for staking. Thus, PoS chains *can only ever have objectively secure fork-rules* when other factors are included in their conversion contexts (like using PoR with a PoW chain, or some system of automatic checkpoints). One thing PoS chains could try is: measuring weight *in another chain's hashes* via PoR.

happens because these conversion methods *don't try to convert work done at different times.* PoR only ever converts *near-simultaneous work*, i.e., if the coin-weights of reflecting blocks are summed, that is always converted to local work *with respect to some specific moment in time.*

While measuring weight in coins (in this case, at least) seems to have some meaning, we probably shouldn't *leave* chain-weight in those units. The difficulty of a PoW network converts network size (participation) into hashes, and it is adjusted regularly. If a chain-weight measurement doesn't account for this, then *how does it include participation at all?* Without including participation in chain-weight, how can two local alternate histories be meaningfully compared? When measuring and converting chain-work, we *always* want to convert confirmations or coins back to meaningful units which factor in *participation* in some way.

2.6 Reflection Between PoW and PoS Chains

Aside

Whether PoS systems *can* be secure is not a focus of this paper. There are still criticisms of PoS without adequate answers. The intention of sections like this is not to endorse PoS, but rather to explore what is possible *if* PoS can be done securely.

Perhaps one of the most interesting features of *Proof of Reflection* is that PoW chains and PoS chains can reflect one another. Up till now, we've contextualized the weight of a reflection via the *work* required to produce a block. But the concept of *work* does not neatly apply to foundational consensus mechanisms that do not require the utilization of some physical resource — such as PoS.

Term

Foundational Consensus Mechanisms: Those mechanisms, like PoW and PoS, which can work in some *standalone* fashion; PoR is a cross-chain *extension* to such mechanisms.

Putting the issue of *conversion* aside for a moment, is it possible *in principle* for PoW and PoS chains to reflect one another? Yes. Additionally, PoR provides decisive advantages *both* for PoW chains *and* PoS chains, though there are some additional problems that must be solved, too.

If a PoW chain is reflected in a PoS chain, then an attacker will likely need more than just computational resources to attack the PoW chain. Consider a PoW chain and a PoS chain that share a root token, and each chain hosts approximately 50% of the total supply. If the two chains have equal block production frequencies, then (using Algorithm 4) 50% of the network's security comes from each chain.

Consider an attack on the PoW chain and presume that the difficulty on the PoW chain is constant over the attack, i.e., the PoW chain's difficulty doesn't adjust quickly enough to react to the attack. Additionally, assume the attacker has *not* been contributing to the network before the attack, i.e., their hash-rate is not accounted for in the PoW chain's difficulty. Given the two chains are mutually reflecting, half of the network's security is provided by the PoS chain (and thus immune to the attacker in this case). Therefore, a successful attacker — *using the traditional method of mining a competing chain-segment in private* — must generate more blocks than both chains combined. That means the attacker needs *twice* the honest hash-rate for a guaranteed successful attack.

However, consider the case that *the security contribution of the PoW chain is capped at 50%* — i.e., capped at the proportion of root tokens hosted on that chain. For our purposes, this situation is approximately equivalent to that where the PoW chain has a *perfect* difficulty adjustment algorithm, i.e., the network instantly adapts to keep the block production frequency constant. For the sake of this demonstration, assume that these chains *retroactively* adjust block weightings to ensure this cap holds. Let $p > 0$ be the honest miners' contribution to *overall* network security, and $q > 0$ be the attacker's contribution. As the PoW contribution to overall security is capped at 50%, the equality $p + q = 0.5$ is enforced. In this case, the attacker will have a maximum chain-weight contribution rate of $\frac{1}{2} \cdot \frac{q}{q+p}$ and the honest chain-segments will have a maximum contribution rate of $\frac{1}{2} \cdot \frac{p}{q+p} + \frac{1}{2}$.

The condition for a successful attack is shown in Equation 25, and the inequality has no solutions.

$$\begin{aligned} \frac{1}{2} \cdot \frac{q}{q+p} &> \frac{1}{2} \cdot \frac{p}{q+p} + \frac{1}{2} \\ q &> p + (q+p) \\ 0 &> 2p \qquad \text{which is a contradiction since } p > 0 \end{aligned} \tag{25}$$

Given the right set-up, a PoW chain gains an *incredible* security advantage from mutual reflection with a PoS chain.

Aside

Note: The attack scenario above assumes that the attacker is not attacking the PoS chain that is reflecting the PoW chain. That is not a safe assumption. Additionally, with traditional blockchains (which are trees), an empty-block DoS is possible — this is addressed in Section 4.8.

What about the PoS chain, though; what benefits does it gain from this relationship? The answer here is simple: by using mutual PoR with a PoW chain, the PoS chain gains *thermodynamic security*; the PoS chain's history is *thermodynamically secured* by the PoW chain. **This solves the *Nothing at Stake* problem for any well constructed PoS scheme.**[30] Furthermore, it is possible for error-correction methods like *slashing* to be implemented *on the PoW chain*, not the PoS chain. Moving the staking and error correction methods to a different chain will require subtle and precise protocol design, but such changes are *in principle* possible with tolerable overhead.

There are some (as yet) unsolved problems that arise through this design, such as the *economic* details of managing block rewards across the PoW and PoS chains. Given that solutions to this problem likely depend on the specific details of the relevant PoS systems, this problem is not addressed here. Note: conversion methods for reflected weight, like Algorithm 6, will work provided a well defined WEIGHTOF function exists.

There are some other conjectured solutions to the *Nothing at Stake* problem.

Quote

> Long-range "nothing-at-stake" attacks are circumvented through a simple "checkpoint" latch which prevents a dangerous chain-reorganisation of more than a particular chain-depth. To ensure newly-syncing clients are not able to be fooled onto the wrong chain, regular "hard forks" will occur (of at most the same period of the validators' bond liquidation) that hard-code recent checkpoint block hashes into clients.
>
> — *Dr. Gavin Wood; Polkadot Whitepaper, s5.2*

Quote

> Provided that stakeholders are frequently online, nothing at stake is taken care of by our analysis of forkable strings (even if the adversary brute-forces all possible strategies to fork the evolving blockchain in the near future, there is none that is viable), and our chain selection rule that instructs players to ignore very deep forks that deviate from the block they received the last time they were online.
>
> — *Ouroboros: A Provably Secure Proof-of-Stake Blockchain Protocol, s10*

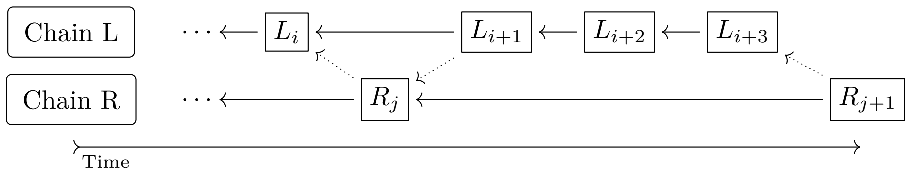
These two examples solve the *Nothing at Stake* problem via mechanisms that are *external* to the protocol itself, i.e., hard-coded checkpoints and the requirement that nodes are online "frequently".

The solution provided by mutual reflection with a PoW blockchain — i.e., thermodynamic security — is provided *by the protocol itself* and can only *increase* the security of PoS mechanisms. Thus, UT's solution to *Nothing at Stake* is qualitatively superior.

[30] I consider the *Nothing at Stake* problem and *long range* attacks to be two sides of the same coin. Maybe it's worth explicitly mentioning that mutual PoR solves long-range attacks, too.

2.7 Counting Work

When chains L and R have the same block frequency and produce blocks in an orderly fashion, it's trivial to count how much work is contributed by each PoR. However, real-world blockchains do not produce blocks like this — even if chains L and R have the same block frequencies, sometimes L will produce 2 or 3 (or more) blocks before R produces its next one (or vice versa).



Should R_{j+1} include 3 PoRs, or just 1? How should R_{j+1} calculate the weight of those PoRs and the total weight contributed to chain R?

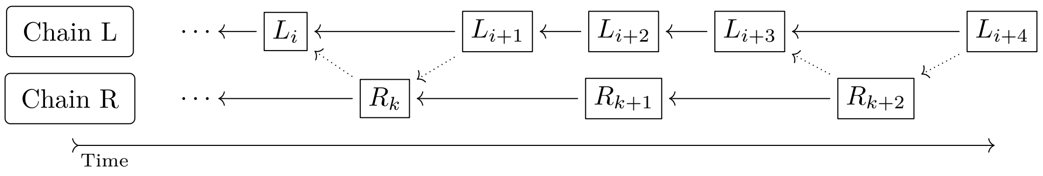
If R_{j+1} had reflected only L_{i+1}, then it's clear that we would count work from both L_{i+1} and R_{j+1} — that much is easy. However, R_{j+1} reflected *three* L headers. Should R_{j+1} count work from *all three*, or something else? Why?

Consider the situation from the miner's perspective (the miner of L_{i+2}). There is no new R block for them to reflect — if there were, they'd include it. The miner is effectively claiming — via the absence of a new reflection — that L_{i+2}'s parent (L_{i+1}) reflects the most recent R header (R_j); that L_{i+1} is still *up-to-date*. From this perspective, it's clear that L_{i+2} should contribute reflected weight to the best R headers that were most recently reflected: in this case, just R_j.

We can evaluate L_{i+3} the same way.

So, yes R_{j+1} should include and count work from all 3 PoRs in this case.

What about a more complicated example?



R_{k+2} can be evaluated by similar logic to R_{j+1} above, but what about L_{i+4}?

Does L_{i+4} count the weight of R_{k+1}? If so, does it count towards L_{i+3}'s chain-weight?

The difficulty in answering these questions — with what we've covered so far — is that *the fork-rule doesn't care* where the weight is added, just that it *is* added. When the fork rule compares L_{i+4} to an alternative block, only the *total* chain-weight of each is relevant. However, to understand exactly how PoRs *should* be counted, we need to know more.

The essence of PoR is *the confirmation of another chain's <u>history</u>* — so a reflection cannot contribute to a block which exists *in the reflecting block's <u>future</u>*. It must be that reflections contribute to the *most recent common ancestor* of all *possible* L blocks which *could* include that specific PoR.

This leads us to a simple and elegant way to count reflected work: follow the arrows back from the reflecting block to the most recently reflected local block.

L_{i+4}'s reflection of R_{k+2} contributes chain-weight to L_{i+3} because *any* block that builds on L_{i+3} could reflect R_{k+2}. Following the arrows: $R_{k+2} \dashrightarrow L_{i+3}$.

Similarly, L_{i+4}'s reflection of R_{k+1} (which could be either explicit or implicit via R_{k+2}) must contribute chain-weight to L_i because *any* block building on L_i could reflect R_{k+1} and include that PoR. Following the arrows: $R_{k+1} \rightarrow R_k \dashrightarrow L_i$.

We can also say more about R_{k+2}'s reflection of L_{i+3}, now, too. Particularly, that the reflected work of $L_{i+1} \cdots L_{i+3}$ should contribute to R_k, *not* R_{k+1}.

3 UT$_i$: Constructing *Ultra Terminum*

Proof of Reflection can be used to build UT$_1$ — an $O(c^2)$ foundation for a blockchain network (called *the simplex*). This section details the construction of such a foundation, and how it can be extended up to UT$_3$ — which has $O(c^4)$ complexity. The $O(n)$ scaling configuration (UT$_\aleph$) is detailed in Section 6.

Such a foundation (*the simplex*) is *not* a sharded blockchain — there's no requirement that participating chains are interchangeable or using the same primitives. This was demonstrated via the example in Section 2.2. Rather, *the simplex* is an emergent construct that is created via the *relationships* between blockchains. Instead of one blockchain being split into many (as occurs with sharding), *the simplex* is many blockchains becoming one coherent network.

3.1 Generalizing Reflection

Proof of Reflection is, in essence, the idea that a chain can acknowledge that its history has been confirmed by a different chain, and that this fact can be used to share security between chains. That is a simplification, but it *is* the essence of it.

In principle, the necessary capabilities (and actions) that some chains, C_A and C_B, must have (and do) in order for C_A to be reflected by C_B are:

1. The headers of C_A can be (and are) freely recorded — promptly and unambiguously — in C_B;
2. The headers of C_B can be (and are) freely recorded — promptly and unambiguously — in C_A; and
3. C_A is able to (and does) promptly prove that its past headers have been recorded in C_B, and has full knowledge of which headers have been recorded.

Algorithm 6 Implementation of ReflectedBlockWeight to support an arbitrary number of reflecting chains.

```
procedure ReflectedBlockWeight($L_i$, $state$)
    $w \leftarrow 0$                                                    ▷ Sum of weights
    for $R$ in AllReflectingChains($state$) do
        $w_R \leftarrow 0$                                              ▷ Sum of converted weights for this $R$
        $n_R \leftarrow 0$                                              ▷ Count of reflecting blocks
        $R_{\text{blocks}} \leftarrow$ ChainHeadsOfVia($R$, $L_i$, $state$)          ▷ $R$ chain-heads known to $L_i$
        for $R_i$ in $R_{\text{blocks}}$ do
            RCHs $\leftarrow$ ReflectedChainHeads($R_i$, $state$)       ▷ Local blocks reflected by $R_i$
            if $L_i$ in RCHs then
                $n_R \leftarrow n_R + 1$
                $w_R \leftarrow w_R +$ WeightOf($R_i$, $state$)          ▷ Convert weight of $R_i$
            end if
        end for
        weightMod $\leftarrow$ CapWorkOf($R$, $n_R$, $state$)           ▷ Optionally cap the work of $R$
        $w \leftarrow w + w_R \cdot$ weightMod
    end for
    return $s$
end procedure               ▷ This is an inefficient method and would not be used in production.

procedure CapWorkOf($R$, $n$, $state$)                                  ▷ A simplistic method for capping work
    $L_f \leftarrow$ BlockFrequencySelf($state$)
    $R_f \leftarrow$ BlockFrequencyOfChain($R$, $state$)
    $e \leftarrow R_f \div L_f$                                          ▷ Expected blocks from $R$ per $L$ block
    return $\min(1, e \div \max(1, n))$                    ▷ Cap the contribution of $R_{\text{blocks}}$ per $L$ block
end procedure
```

The benefits from *Proof of Reflection* begin as soon as C_A integrates this knowledge into its chain-weighting algorithm, by a method suitably similar to Algorithm 2 and Algorithm 3.

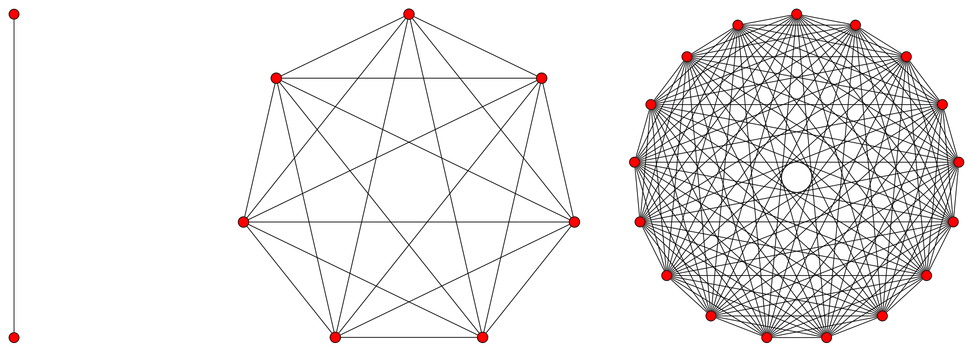
If C_A and C_B are doing *mutual* Proof of Reflection, then both chains must satisfy all requirements.

Is C_A able to *simultaneously* do reflection with more than one other chain, e.g., $C_C ... C_Z$? Yes. There is nothing that we have covered so far that would prevent this. If *Proof of Reflection* is viable with a single other chain, then it is viable with *many* other chains. However, the dynamics do become increasingly complex, as we will soon see.

In order to support arbitrarily many reflections, we need to modify ReflectedBlockWeight from Algorithm 2 as shown in Algorithm 6.

Note that Algorithm 6 integrates the cap on weight contributed by each reflecting chain, as suggested in Section 2.6.

3.2 UT$_1$: The Simplex



(a) Proof of Reflection between 2 blockchains. A 1-simplex. The most basic non-trivial simplex.

(b) Proof of Reflection between 7 blockchains; a 7-chain simplex; a 6-simplex.

(c) A 17-chain simplex; a 16-simplex. It has 136 unique mutual reflections in total.

Figure 12: Simplexes of increasing capacity. Vertices are simplex-chains. Edges are the reflections between simplex-chains.

Term

Simplex: The single coherent structure that emerges from a collection of blockchains that mutually reflect each other.

When two or more blockchains *mutually reflect* each other, they form a *simplex*[31]. For the sake of brevity: all *reflections* within a simplex are *mutual reflections*, and I will omit *mutual* from now on when discussing them. Examples of simplexes are shown in Figure 12.

When a blockchain is part of a simplex, it is called a *simplex-chain* (as distinct from *dapp-chains*).

Term

Simplex-chain: A blockchain that is part of a *simplex*; it mutually reflects all other simplex-chains in that simplex.

To maintain consistency with the geometric usage of the term *simplex*: a simplex with $k+1$ chains is called a k-simplex or a $(k+1)$-chain simplex[32]. In a k-simplex, each simplex-chain has k reflections

[31] The name is taken from geometry (particularly: the higher-dimensional kind). A simplex, for a given dimensionality, is the uniquely simplest polytope; e.g., a line in 1D space, a triangle in 2D space, a tetrahedron in 3D space, etc. A k-dimensional simplex is known as a k-simplex. As shown in Figure 12, a particular 2D projection of a k-simplex (which produces a regular $(k+1)$-gon with additional edges between all pairs of vertices), is identical to a diagram of all possible mutual reflections between $k+1$ blockchains, where each chain is represented by a vertex and each mutual reflection is represented by an edge.

[32] NB: I will ignore this distinction for $k \gg 1$.

(one reflection for each of the other simplex-chains). A k-simplex has, in total, $\binom{k+1}{2}$ reflections.

The security of simplexes is discussed in Section 4.10.1.

3.3 UT$_1$: Scaling Complexity Intuition

Aside Although we will rigorously investigate UT's scaling complexity in Section 5, it will prove useful to introduce N_1 here and to gain an intuition for UT's scalability before Section 4.

How scalable is UT$_1$?

Let us start by assuming that a single blockchain has $O(c)$ capacity (measured in transactions per second) and is thus $O(c)$ scalable. We'll also assume that the *headers* are $O(1)$ bytes (i.e., a constant size).

Note that a single blockchain is a 0-simplex.

A 1-simplex has two blockchains, and each chain must record the headers of the other and the respective PoRs. For the sake of simplicity, let's assume that the PoRs are also $O(1)$. Thus, each chain loses $O(1)$ of its $O(c)$ capacity, and the network's total capacity is $2O(c) - 2O(1) = O(c)$.

Since each chain has finite capacity, there is an absolute maximum size to a simplex. We can keep adding more chains until all blocks are 100% full of headers and PoRs. However, this system has 0 capacity for transactions. So, there must be a capacity maxima with more than 2 simplex-chains and fewer than the absolute maximum.

What we're observing is that the overall capacity is the *product* of the size of the simplex (in chains) and each chain's remaining capacity — a quadratic relationship.

Let's call the proportion of capacity for transactions t, and the proportion for reflections r. We know that $t + r = 1$, since $t + r$ is 100% of a chain's capacity. However, the *network-wide* capacity is proportional to the *product* of t and r. Maximizing tr suggests that a chain's capacity should be split approximately 50/50 between t and r, so $t = r = 1/2 = O(1)$. Given that t and r are proportions of the capacity, we can say $O(ct) + O(cr) = O(c)$. Thus, the overall capacity is given by $O(ct) \cdot O(cr) = O(c^2)$.

This relationship is analogous to the relationship between a rectangle's maximum area for a given perimeter length: the area scales with the square of the perimeter (and such a rectangle is a square).

Let's say N_1 is the number of chains which maximizes a simplex's capacity (for transactions). To get a feel for N_1, let's estimate it using example chains based on modified Bitcoin parameters.

First, we'll add a merkle root of headers and PoRs to the header, increasing header size from 80 to 112 bytes. Second, we'll reduce the block frequency down from 10 minutes to 1 minute, and scale the block limit down from 1 MB to 100 kB (this will give us more realistic numbers). Let's assume the merkle branch leading to a header is 256 bytes, so the total PoR overhead is 368 B / block / chain. We want to use 50% of the overall capacity (100 kB / block) for PoRs, so each block should contain ∼50 kB worth of PoRs. Dividing the PoR capacity by the per-chain overhead gives us $N_1 \approx 139$ chains and a network-wide capacity equivalent to ∼70 Bitcoin networks.

3.4 UT$_2$: Dapp-chains

Dapp-chains are a method by which *Ultra Terminum* exceeds $O(c^2)$ scaling *without* using the method described in Section 6. To be clear: the $O(c^2)$ configuration of UT is compatible with that other method; dapp-chains are a *separate and independent* method of scaling. However, there are *decisive* reasons to introduce and use *dapp-chains*. Dapp-chains provide features that the $O(n)$ scaling configuration alone cannot *easily* provide. Additionally, dapp-chains increase the simplex's scalability to $O(c^3)$ or $O(c^4)$.

Term

Dapp-chain: An *application-specific* child-chain that is secured via the parent-chain. Dapp-chains may have architecturally distinct state- and transaction-schemes (distinct from those schemes used in the simplex, and other dapp-chains).

Intrinsically, dapp-chains are not restricted to any particular foundational consensus method. They might use PoW, or PoS, or PoA, or something else. However, dapp-chains also use *Proof of Reflection* with their host simplex-chain. With a suitable foundational consensus method, PoR enables dapp-chains to be as secure as their host simplex-chain with little overhead. Note that, since dapp-chains can use whichever foundational consensus method, they can optionally have *their own* root token (and use that for mining rewards, transaction fees, etc).

It's preferable that a simplex-chain validate the headers of its dapp-chains (similar to a light client), though this is not required. For some dapp-chain consensus methods (such as PoS), there might be special primitives that a host simplex-chain must support. However, only that host simplex-chain requires those primitives; other simplex-chains do not.[33] This means that simplex-chains can *specialize* in hosting *particular types* of dapp-chains, providing rich and efficient environments (for nodes of both simplex-chains *and* dapp-chains).

Validating dapp-chain headers, on-chain, can be done via the following simple, clean, and extensible method: *encode dapp-chain headers as simplex-level transactions*. This means that supporting new dapp-chain consensus methods is about as difficult as introducing new transaction types (or opcodes), and different simplex-chains have a great deal of freedom in choosing which dapp-chain consensus methods to support.

Term

Header-transactions: Dapp-chain headers that are encoded as simplex-level transactions; i.e., they are processed by a simplex-chain as a transaction, but they also function as the header for a dapp-chain block.

Practically speaking, a simple input-output transaction system with light scripting capabilities (like that of Bitcoin) can be created to facilitate the necessary primitives. Additionally, different simplex-chains can implement different scripting systems, effectively facilitating *any* compatible consensus mechanism. There is not much (if any) overhead to using an input-output system like this: a header's parent hash is like a transaction input, the *output* is the header or its hash[34], and any other particulars of the header can be treated as an input script to the transaction[35].

3.4.1 Dapp-chain Security

If the headers of dapp-chains are simplex-level transactions, what can we say about the security of dapp-chains?

First, notice that there is no substantive difference between standalone headers and header-transactions. That means that *zero-confirmation* header-transactions are *exactly* as secure as a standalone counterparts (and at least as secure as zero-confirmation transactions). This is not very secure in the case of a PoW dapp-chain, but it means that a PoS dapp-chain's zero-confirmation header-transactions could be just as secure as blocks from an equivalent standalone PoS blockchain. (It also means that the PoS dapp-chain is more secure after header-transactions are confirmed, compared to the standalone equivalent.)

When a header-transaction is confirmed by the simplex, the corresponding dapp-chain can efficiently

[33]The caveat here is that other simplex-chains may need to be capable of validating fraud proofs for the simplex-chain in question. So they don't need these primitives *available to local transactions*, but do need to be capable of executing those primitives if a fraud proof involves one.

[34]A header-transaction's output script can be generic (or templated) as it is the same for all header-transactions for that dapp-chain. In practice this can be as simple as a single opcode that validates that header. In Bitcoin, an output script is known as the `scriptPubKey`.

[35]In Bitcoin, the input script to a transaction is called the `scriptSig`; see https://en.bitcoin.it/wiki/Transaction.

use one-way PoR to partially[36] inherit the security (and security properties) of the host simplex-chain.[37] Similar to mutual PoR, this can provide a *security context* where otherwise-insecure methods of consensus can be done securely.

With regards to doublespends, one-way PoR means that the reflected chain is *at least* as difficult to attack as the reflecting chain (as we covered in Section 2.2.4). Since the parent simplex-chain is as difficult to attack as the complete simplex, each dapp-chain must therefore *also* be that difficult to attack. Provided that a dapp-chain's blocks are available, attacking a dapp-chain is as difficult as attacking the entire network.

Note that parent-chains (generally) need to record their child-chains' headers *anyway*, so this use of one-way PoR — where a simplex-chain reflects child dapp-chains — has near-zero overhead for both the simplex-chain and the dapp-chain.

One major, generic concern for dapp-chains is *preventing DoS attacks.*[38] This is one reason to favor PoS (or PoA) dapp-chains over PoW dapp-chains. Another concern is the *availability* of dapp-chain blocks.

3.4.1.1 Spam, Availability, and Dapp-chains: A PoW/PoS Asymmetry

PoW Dapp-chains Proof of Work systems operate best when all practicable hashing resources are contributing to the same system. That is: all economically available miners are mining on the same network.

We can explore this with a thought experiment. Say I forked the Bitcoin codebase and created a new genesis block with today's date, which in turn creates a new blockchain network. Crucially, I do not change the hashing algorithm.

How secure would this network be? Given that the actual Bitcoin network has an established hash-base, and there are many old, uneconomical mining units out there, the overwhelming majority of hash power would not be working on my new network (either it would be off, or working on Bitcoin). In fact, even if all the units that were powered off were used to mine my network, it would still have but a small fraction of Bitcoin's hash-rate. The effect of this is that *at any point* a small proportion of Bitcoin miners could divert their miners and perform a 51% attack against my network. Therefore, *given Bitcoin exists,* my network cannot be considered secure in practice.

If two traditional PoW chains share a hashing algorithm, then the one with more hashing power is the only secure candidate.

How does this apply to PoW simplex dapp-chains? First, notice that if a PoW dapp-chain has a comparable difficulty to a simplex-chain, then the simplex as a whole (including other dapp-chains) would benefit if that hashing power were used on the simplex instead of the dapp-chain. Additionally, moving hashing power the other way (from simplex to dapp-chain) would weaken the overall security of the simplex. From this, we can intuit that the most stable, secure system is one that heavily prioritizes mining PoW simplex-chains over PoW dapp-chains. This brings us to our second observation: that PoW dapp-chains will naturally have a much lower difficulty than their host simplex-chains.

In a functioning PoW network, the work serves not only as the method of securing the chain, but also a method to prevent block spam. This is due to the DAA regulating block production rates. DAAs are unreliable, though, when there are large reserves of mining power that might come online at any moment for an unknown period of time, and then disappear. One typical consequence of

[36]Some security properties of the host simplex-chain are not inherited. For example: the simplex ensures data availability of simplex-chain blocks, but not dapp-chain blocks. Other security properties, like thermodynamic security, are inherited.

[37]Note: PoW dapp-chains will have a much lower difficulty than the host simplex-chain. Although a simplex-chain could do mutual PoR with dapp-chains, this is unnecessary and inefficient — provided that this difficulty asymmetry exists. Although there is no fundamental reason that PoW dapp-chains must have a much lower difficulty, we should take care to avoid any implementation that would compromise or reduce the security of the simplex. Practically, this probably means avoiding PoW dapp-chains (see Section 3.4.1.1).

[38]Unfortunately, the strategy we use in Section 4.8.3 to protect simplex-chains does not work here.

a sharp drop in mining power is a period where the difficulty is much higher than the remaining miners are capable of meeting, resulting in reduced block production. In extreme cases, this may require a hard-fork to resolve. The usual solution to this is a rule whereby blocks at lower difficulty can be mined, provided that a suitably long duration has passed since the prior block. If PoW dapp-chain headers are verified by the simplex, a hard-fork to change the DAA of a PoW dapp-chain would require the host simplex-chain to be aware of this change. This is not the only issue with block spam, though. If a miner can trivially create blocks (because the difficulty is too low), then they could flood the host simplex-chain with valid (but useless) header-transactions. This is a problem, not just for that dapp-chain, but for all other dapp-chains hosted by that simplex-chain.

Aside These are substantial problems, but might be soluble. The next problem, however, is not.

Spam is not the nail in the coffin for PoW dapp-chains. Rather, that honor falls to *availability*.

The problem itself is simple to describe: a malicious miner creates valid blocks, and publishes the header-transactions to the host chain, but does not publish the block bodies to the dapp-chain network.

A key requirement of dapp-chains is that they cannot increase the load of the miner above $O(c)$. This means that processing a single header-transaction cannot exceed $O(c/N_D)$, where N_D is the number of dapp-chains that a simplex-chain is capable of hosting – or, more accurately, the number of header transactions it can process per block; we'll assume these are equivalent. Ideally $O(N_D) = O(c)$, assuming it is not possible to do better than $O(1)$ when processing a header-transaction.

If we are to have reliable inter-chain operation between simplex-chains and dapp-chains, then we must have some assurance that the recorded blocks are valid, which requires them to be available. Invalid blocks would allow a miner to steal funds via cross-chain transactions, which is obviously bad. Provided that we only allow (public[39]) dapp-chains which support fraud proofs, then we have the foundation for that assurance. But, fraud proofs only work if honest nodes have access to the block to calculate said fraud proof. It is generally not possible to create a fraud proof if the block is unavailable.

Since availability is a requirement, how can we ensure that dapp-chain blocks are available and not withheld?

Aside Note that *Proofs of Availability* are of no use here, because verifying them requires at least as many bytes as the block[a] itself; it is just as bandwidth intensive as downloading all blocks. So, at best, we might be able to do some kind of challenge-response, but, practically speaking, this isn't the solution we're looking for.

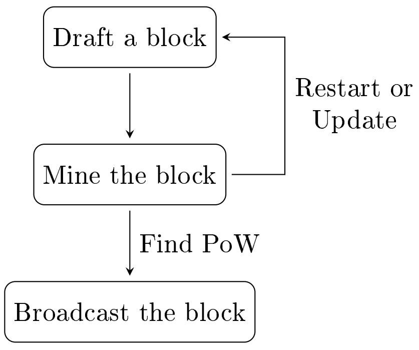
[a]This is somewhat intuitive since proofs of availability allow for the reconstruction of the original block.

I suspect that, for PoW dapp-chains, it is *in principle* not possible to ensure the availability of blocks. This is due to the principles by which PoW blockchains operate. Particularly, *proofs of work* are created *in isolation*. Only *after* a PoW is found is a block shared with the network (see Figure 13a). Therefore, it is always possible to partially share a block in (pure) PoW networks.

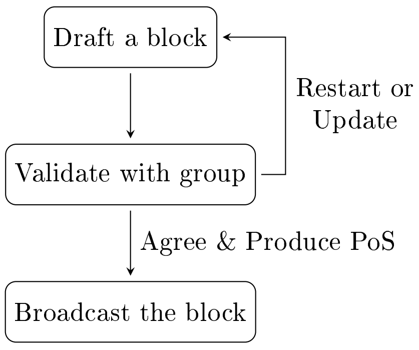
PoS Dapp-chains First, it's important to note that there are many variants within the "Proof of Stake" family of consensus methods. For our purposes right now, we're interested in those where multiple validators are involved before a block is finalized / validated. Figure 13b demonstrates a simplification of such a block creation processes.

Provided that the PoS method is well-constructed, this kind of PoS solves both the spam and availability problems.

[39]If a dapp-chain is designed to be private, opaque, or non-financial, then we might not care (or even be able to care) about the validity of its state transition.



(a) The PoW block creation process (simplified).



(b) An example of the kind of PoS block creation process we're after.

Figure 13: The block creation process (simplified) for PoW chains and some variants of PoS chains.

Typically, such systems prevent validators from creating competing blocks by imposing an economic penalty (*slashing*) on those validators who help create two or more blocks at the same height (or slot, or whatever). Since slashing will destroy a validator's capital, and thus destroy their ability to be a validator, we can infer that this process will be rare. That's good, because we don't have to care much about the overhead of resolving these cases. This solves the spam problem.

The availability problem is solved provided there is at least one honest validator with knowledge of the block contents (because honest validators want to broadcast the block).

Aside

The term "well-constructed" is doing most of the heavy lifting here. I don't think we need to worry, though, because these sorts of problems are what PoS designers spend a lot of time on. So, if any contemporary PoS chains are secure, then we *should* have a viable option for PoS dapp-chains.

PoA Dapp-chains & Others Regarding spam and availability — what about PoA? Fortunately this case is trivial: we offload the responsibility *completely* to the PoA operators. The reason is simple: if PoA operators are withholding blocks or committing fraud, then that dapp-chain is already compromised.

And to cover all our bases: if interoperability with the simplex is not required, then we don't care what consensus method is used; the simplex-chain is just time-stamping data.

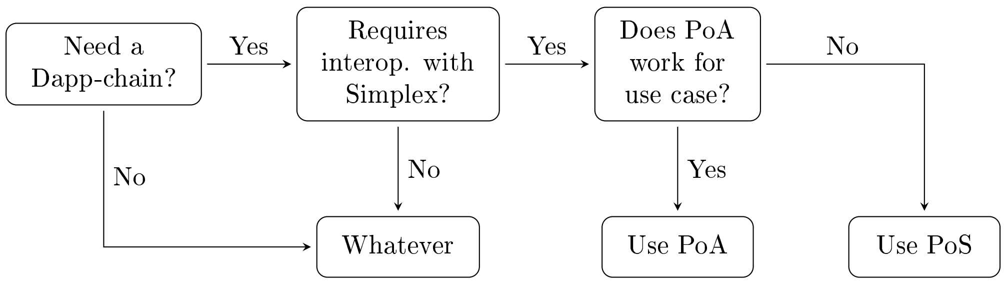


Figure 14: A flow chart to decide which proof system to use for a dapp-chain, assuming a secure PoS method is available.

It is worth noting that there are other directions that dapp-chains could take, too. Notably: we could use some kind of rollup; we could use some kind of zero-knowledge proof (ZKP) system; or we could go in a more traditional layer-2 direction, such as state channels.

Design Position With regards to how we should actually design simplex-chains to best support dapp-chains, there are a few important takeaways from the above. First, different dapp-chain use-cases have different technical requirements; we don't need a one-size-fits-all design. Second, no PoS system is yet adapted for dapp-chain use, and if PoS is generally insecure then we cannot assume secure PoS dapp-chains are possible. Third, scaling research (in a broad sense) is often focused on methods that should be compatible with our dapp-chain design, so new methods may become available at any point. Therefore, we should aim to support many different methods of running dapp-chains and providing the necessary primitives for users to do so.

3.4.2 Three General Incentive Models for Dapp-chain Reflection

If dapp-chain headers are included alongside transactions in simplex-blocks, is it not the case that both must pay some kind of *transaction fee*? If not, how are simplex-chain miners to prioritize what to include in their blocks? Even if such a fee is *not always necessary*, the *ability* to provide a fee has decisive advantages — like creating asymmetry between an attacker and honest simplex-chain miners.

If it is possible to implement dapp-chains (or any system of child-chains) such that those chains have *freedom of protocol* and *freedom of incentivization* whilst inheriting the parent-chain's security, then we should strive to achieve that.

Term

Freedom of Incentivization: The property whereby child-chains have free choice of incentive-system (i.e., the nature and dynamics of their root token, or lack thereof).

Term

Freedom of Protocol: The property whereby child-chains have free choice of protocol (including consensus mechanism, scripting, accounting methods, block structures, etc).

Section 2.4.2 details a conversion method whereby PoR is possible between chains using different root tokens via a DEX. Could dapp-chains use a *protocol-level* DEX to abstract their protocol and incentive method away from those of its parent-chain? Yes.

Is this required for this kind of abstraction? No.

Here are three methods of abstraction which maintain the above freedoms.

3.4.2.1 Method 1: Pay the simplex miner on the dapp-chain

In this method, the dapp-chain uses its root token to pay both the dapp-chain miner and the simplex-chain miner (who includes the relevant dapp-chain header in their block).

Since all dapp-chain miners are required to run a full node of the parent-chain, this is trivial. In essence, the host simplex-chain is a subset of the dapp-chain. Simplex-miners can run light clients[40] of the dapp-chain to regularly collect block-rewards.

A dapp-chain could, perhaps, have a rule like *X root tokens are created as part of the coinbase transaction and the miner of that dapp-block has free choice of the proportion of those which are provided as a transaction fee to the host-miner.*

Example use-case: an existing blockchain migrates to become an *Amaroo* dapp-chain.

3.4.2.2 Method 2: Pay the simplex miner via a native DEX

When a dapp-chain hosts a native DEX, it can use that DEX for PoR. The general case (where a reflecting chain contributes far more chain-work than the reflected chain) was discussed in Section 2.4.2.

[40] A simplex-miner could use other methods too, like maintaining full nodes of each dapp-chain and continuously cycling through them (alternating which are running and which are not) to avoid massive computation requirements. Light clients seem obviously preferable where possible.

Consider the limited context of a DEX with only one required trading pair (between the dapp-chain's root token and the ROO), combined with the security-contribution differential between a simplex-chain and a dapp-chain. Note that a conservative implementation of a DEX between this pair *only relies on local state* — that of the host simplex-chain and the dapp-chain, all of which is accessible to dapp-chain full nodes. The simplest method of preventing market manipulation (that might allow for some attack on the dapp-chain) is to calculate PoR weight via an *old* exchange rate (e.g., from 24 hours ago), or to use an *average* over some period of time. Both of these ensure that *competition between blocks* (at any given time) is not dependent on the *current* DEX execution. With regards to dapp-chains using Proof of Reflection, this is sufficient.

Given a DEX, the dapp-chain can use this to automatically convert some of the mining reward to the root token of the host simplex-chain. These rewards could accrue over time and be bundled into far fewer transactions than would otherwise be necessary and automatically managed by the DEX. Unlike the previous method, this method doesn't require the simplex-chain miner to ever interact with dapp-chains (besides including their header-transactions); however, the protocol is more complex.

Example use-case: a greenfield dapp-chain uses an Amaroo-compatible DEX (which requires no development effort) so that simplex-miners have lower operating costs; thus incenting simplex-miners to include their headers over those of others.

3.4.2.3 Method 3: Pay the simplex miner directly

If the dapp-chain is willing to forego more efficient SPV transactions (or otherwise doesn't require them), and it is willing to bear the full burden of PoR in this context, then simply recording the hash of a dapp-chain header might be sufficient. In such a case, transactions (in the style of Bitcoin's OP_RETURN transaction format) provide everything required. This makes sense if the dapp-chain has exceptionally large headers, or if the dapp-chain does not wish to *disclose* the headers themselves (perhaps it is a private/permissioned network). In any case, since it is impossible to stop users including hashes in transactions, this is always a method by which dapp-chains can enable PoR with the host simplex-chain.

Example use-cases:

- An existing *anchored*[41] blockchain migrates to become an Amaroo dapp-chain.
- A new (and ephemeral) dapp-chain is created to facilitate a national election that will result in a 200 GB audit log (facilitating unprivileged verification of the election result) and a peak votes-per-second over 10^5. This demonstrates both *freedom of incentivization* (as there is none) and *freedom of protocol* as no payments are made and no restriction is placed on the nature of this dapp-chain's payload.

3.4.3 PoS Dapp-chains

If the headers of dapp-chains are encoded as simplex-transactions, then techniques like *slashing* can be first-class operations within the *hybrid* PoW context provided by the simplex. This solves the *Nothing at Stake* and *long-range* problems for PoS dapp-chains, so long as the necessary PoS primitives can be encoded in a simplex-transaction.

The abstraction layer between simplex-chains and dapp-chains brings practical benefits, too. For example: existing (open-source) PoS blockchain schemes can be easily integrated as dapp-chains. Given that dapp-chains inherit security properties of their parent-chain (via one-way PoR), if such a dapp-chain's consensus method supports *other* features — e.g., finality guarantees — we should get those features for *free*. (Sharding, too, for that matter...)

[41] **Anchoring**: The process by which the hash of some data (perhaps a secondary chain's blocks) is included in transactions of a primary blockchain (e.g., Bitcoin). Anchoring *would* be a progenitor to PoR, except that I believe the idea of an on-chain light client predates the term *anchoring*. Though I think that the idea of time-stamping a hash (e.g., via an OP_RETURN transaction on Bitcoin) predates the idea of an on-chain light client.

The most likely method of integration has three core components: modification of the headers (and integration of PoR), modification of existing slashing protocols, and support for intra-simplex SPV proofs. For example: OpenEthereum (Parity; 2022) [42] could be integrated as a dapp-chain with the creation of a new header format, the creation or modification of a suitable engine, and the implementation of suitable builtins that facilitate intra-simplex SPV proofs[43] and any other useful features. Naturally, there are some other components that are necessary (like a component for broadcasting header-transactions), but those components are common over many dapp-chain integrations and only need to be written once for each *kind* of dapp-chain.

3.4.4 Going Further

There are two more ideas we'll explore a little in this section: dapp-chain simplexes and dapp-dapp-chains. Both of these are somewhat speculative, in the sense that we're adding more and more assumptions and should thus be somewhat conservative in our conclusions. However, if these techniques can be done practically and securely, then they will give us some nice features.

3.4.4.1 Dapp-chain Simplexes

Say we have a simplex with a reasonable number of dapp-chains — enough that we're interested in increasing the efficiency of communication between them. Currently, the worst case situation for moving something (tokens, information, permissions, whatever) from one dapp-chain (D_α hosted on L) to another (D_β on R) involves at least three transactions: $D_\alpha \to L \to R \to D_\beta$. If we are just proving something to D_β, we can (at least) do this in one transaction, but we still need to prove both that some block on D_α is valid and that some state entries have some particular values. (The proofs for L and R blocks, and the relevant PoRs, are left as implicit since all full nodes have those already.)

If a path between two dapp-chains is common enough, is there any way to make this process more efficient? Yes: dapp-chain simplexes.

The simpler case is when all the dapp-chains are on the same simplex-chain. In this case, the dapp-level simplex's PoRs can be verified by the host simplex-chain, and we should gain some of the properties that the main simplex enjoys.[44] Participating dapp-chains would know the chain-tips for all the other participating dapp-chains, so SPV-esq proofs between those chains are roughly halved in size.

The more complex case is a dapp-chain simplex where the dapp-chains are hosted on different simplex-chains. Enforcing this at the simplex-level would require some additional protocol support — the host chain would need to verify something that depends on another simplex-chain. As we'll see in Section 4 (particularly Section 4.11), this kind of thing can be done without breaking $O(c)$ constraints. This does not, however, mean that we can have an unlimited number of dapp-chain simplexes that span multiple simplex chains. If we tried that, we'd find that a fully validating simplex-chain node would need to follow $O(c)$ many dapp-chain simplexes, each of which has $O(c)$ many chains, resulting in an overall load of $O(c^2)$.

So, in conclusion, dapp-chain simplexes are possibly useful but should be considered on a case-by-case basis until we have a better understanding of them.

3.4.4.2 UT$_3$: Dapp-dapp-chains

Let's assume for a moment that there exists a feature-complete, production-ready, open source, sharded PoS blockchain. It's out there, in the wild, chugging away processing $O(c^2)$ many transactions.

[42] OpenEthereum itself has, since this section was written, been deprecated. This section is left unchanged since the specific client we might modify is immaterial to the main point: that relatively few modifications can be applied to existing clients to create a dapp-chain version of that kind of blockchain.

[43] Note: instead of builtins, these requirements could be met via EVM/WASM smart contracts.

[44] These properties (of the main simplex) are explored in-depth in Section 4. Exactly which properties are gained in which contexts will be left for future research, but I'd expect there to be benefits around censorship resistance, reduced proof sizes, DoS mitigation, and more.

Given that we just discussed how existing blockchain clients can be modified to run as dapp-chains on the simplex, what, *in principle*, is stopping us from using the same techniques with this sharded PoS chain? If the answer is *nothing*, then it's self-evident that this is a path to take UT from $O(c^3)$ capacity to $O(c^4)$ capacity — that is, $\mathrm{UT}_2 \rightarrow \mathrm{UT}_3$. There may, of course, be more adjustments that need to be made depending on the PoS protocol. But, if the protocol is secure in isolation, then it's hard to see how adding thermodynamic security via one-way PoR with a simplex-chain would *necessarily* be somehow incompatible or problematic. In general, if some $O(c^x)$ scaling method works for a traditional blockchain, then combining it with UT results in an $O(c^{2+x})$ scalable network.

It's worth noting that this technique only works in one direction: UT can accommodate other chains (as dapp-chains), but other chains cannot accommodate UT in the same way. If a sharded chain were adapted such that each shard were part of a simplex, then either overall capacity decreases due to PoR overhead, or the chain hosting the shards is redundant and we just end up with an implementation of UT. This is yet another asymmetry between UT and traditional blockchains.

4 Practical Considerations for UT's Design

4.1 The Availability of Reflected Blocks

What happens if a chain reflects a valid block that is not available? Could an attacker use this to their advantage?

For example, consider the following situation (Figure 15). L and R are two mutually reflecting chains, and the most recently produced block is L_i, which reflected R_k. Next, a malicious R miner produces a valid block (R_{k+1a}) which reflects L_i. That miner broadcasts the *header* of R_{k+1a} along with a PoR branch proving that L_i was reflected, but does not broadcast the rest of the block. The L miners receive the header and PoR branch, and shortly thereafter an L miner produces L_{i+1}, which reflects R_{k+1a}. Since other (honest) R miners cannot build on R_{k+1a}, they must work on the draft block R_{k+1} instead. Note that, although R_{k+1} will almost certainly reflect L_i, whether it reflects L_{i+1} is at the miner's discretion.

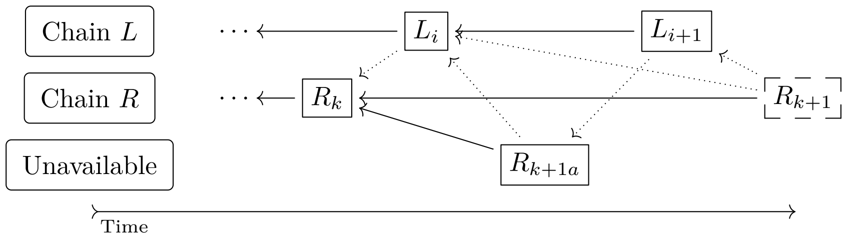


Figure 15: Chain-segments showing an unavailable R block that was reflected anyway. The dashed frame of R_{k+1} indicates that it is a draft block (i.e., not yet mined, but being actively worked on by miners). Note: R_{k+1} reflects L_{i+1} at the miner's discretion — it is optional.

The attacker can use this situation to their advantage if R_{k+1a} has precedence over R_{k+1} (i.e., it's favored by the fork-rule). In that case, the attacker can publish R_{k+1a} *after* R_{k+1} is mined for an advantage.[45] When does this case occur?

If R_{k+1} does *not* reflect L_{i+1}, then R_{k+1} will claim the same chain-weight as R_{k+1a}. However, R_{k+1a} is favored over R_{k+1} since the PoR via L_{i+1} contributes weight to R_{k+1a} (as discussed in Section 2.7). So the attacker *always* has the advantage if R_{k+1} *does not* reflect L_{i+1}.

If R_{k+1} does reflect L_{i+1}, then R_{k+1a} is implied as an earlier block, so should have precedence.

Both these situations make sense: sometimes two R blocks will be created near-simultaneously, and occasionally only one of those blocks will be reflected by an L block. In those naturally arising cases, we'd *expect* the reflected block (R_{k+1a} in this case) to take precedence. Either because it was reflected when R_{k+1} wasn't, or because, for R_{k+1} to reflect L_{i+1}, R_{k+1a} must have existed first.

So, we seem to have a problem: on the one hand, an attacker could withhold a block (but not the header or PoRs) to gain some advantage; on the other hand, prioritizing the earlier block seems to be necessary and correct behavior when this situation arises naturally.

The only difference between these two situations is whether R_{k+1a} is available or not; so this will be our discriminating factor.

Can R miners do anything to prevent L miners reflecting R_{k+1a} before the block body is released? The R miners can't prevent a withheld R block from being produced or reflected, but, R miners can still influence L miners' choice to reflect R_{k+1a} (and withheld blocks in general). L miners want their blocks to be reflected by R miners so that future L blocks will build on top of theirs. So R miners could refuse to reflect some L blocks — particularly, L blocks that reflect R block headers without a known body (such as R_{k+1a}).

[45] This is similar to the *selfish mining* attack published in Majority is not Enough: Bitcoin Mining is Vulnerable (Eyal, Sirer; 2013). It could also be chained to effect a doublespend.

Since L miners face the same problem (withheld but reflected L blocks), both L and R miners are incentivized to cooperate. This forms a stable equilibrium provided that the honest miners have excess bandwidth and that the attacker has less than half the combined (post-conversion) hash-rate.

Another way to look at this is that PoR only works when all blocks are available. This isn't surprising because blockchains generally only work when all (canonical) blocks are available. Since miners only want to participate in systems that work (how else would they earn income?), they will adopt the practices required to make the system work.

4.1.1 The Axiom of Availability

To codify this as a foundation of UT, we will introduce the first network *axiom*: the Axiom of Availability.

Axiom

Axiom of Availability

Principle: All reflected blocks must be verifiably complete and readily available.

Predicate: A block, L_i, cannot yet be valid unless, for each reflected block R_k:
- i. R_k has been downloaded (i.e., the block is available); and
- ii. the cryptographic integrity of R_k has been verified; and
- iii. R_k is valid under this axiom.

Term

Axiom: *Network axioms* are foundational rules expressed as a principle and predicate. Consensus-forming nodes must adhere to them. See definitions: Availability, Maximal Reflection, Unified Ancestry.

4.1.2 Bandwidth Requirements

In the case of just two chains, the bandwidth requirements for L miners are approximately doubled, but the storage and computational requirements are not. While there is some storage and computational overhead on the part of reflected R blocks, these are close-to-negligible when compared to the mined chains: miners need only to store *recent* R blocks and verify that they are *structurally* valid. While this is technically an $O(c)$ computational load per block (since blocks are $O(c)$ bytes large), practically this load is far smaller than the $O(c)$ load of *fully processing* a block, so we can treat it as a constant overhead ($O(1)$). Since only recent blocks are stored, the storage requirements are also $O(1)$.

If there are $O(c)$ many chains, then requirements are: $O(c^2)$ for bandwidth, $O(c)$ for storage, and $O(c)$ for computation. As we will see in Section 5.8, the practical bandwidth requirements are acceptable (on the order of 1 MB/s or so[46]). Assuming that holds true, we can conclude that the Axiom of Availability is not a bottleneck for scalability.

See Section 5.8 for a deeper discussion of bandwidth complexity.

4.2 Proving Reflection

If simplex-chains' consensus protocols require accounting for reflected work, then nodes must have some method whereby they know which work (in a particular chain's history) has been reflected. That is: a node for chain L must be able to answer the question *For each other simplex-chain, which blocks in chain L's history have been reflected?* This means that each node must have $N_1 - 1$ answers, per block, for a simplex of N_1 chains.

There is a trivial method: with each header, include the corresponding merkle branch which proves reflection. Specifically: when a miner on chain L mines a block that includes a header from chain

[46]A rule of thumb: 1 MB/s here corresponds to 1,000-4,000 TPS. Also note that this is only a requirement for miners and full nodes *after* a desired chain has been fully synchronized.

R, they should also include — alongside the header — a merkle branch that shows the most recent chain L ancestor that has been reflected by chain R. For example, block $B_{L,i+1}$ might include a proof that $H_{L,i}$ was reflected by $B_{R,j}$. That branch is the only required branch (i.e., the *missing* branch), as chain L nodes are *already* aware whether $H_{R,j}$ was reflected by $B_{L,i+1}$.

Miners would need to do this for *all* simplex-chains that they reflect. Predictably, this has overhead with order $O(N_1 \cdot \log_2 N_1)$, where N_1 is the number of chains in the simplex. This method has complexity $O(c \cdot \log_2 c)$ which is discussed in Section 5.6.2.

Term

Explicit Proofs (+PoRs): The UT protocol variant wherein miners/validators explicitly record *both* reflected headers *and* the single missing merkle branch required to prove reflection.

Do we *need* to include proofs of reflection, though? Is it possible to avoid the explicit inclusion of those proofs, potentially allowing for $O(c)$ complexity instead?

If miners of any simplex-chain download the blocks of *all* simplex-chains — as mentioned in Section 4.1 — then including all necessary proofs of reflection can be made redundant. Since miners, theoretically, have all the necessary data to construct the proofs, do those miners need to actually include those proofs? Could we treat those proofs as witnesses and prune them — similar to Segregated Witness (SegWit)[47]?

Term

Omitted Proofs (+OP): The UT protocol variant wherein miners/validators explicitly transmit *only* the reflected header component of PoRs, such that necessary proofs of reflection themselves are deterministically recalculable.

There would be some downsides to omitting the proofs of reflection. For one, it would mean that simplex-chain nodes of a single chain, during an initial sync, would not be able to verify the PoRs without auxiliary data — potentially a lot. Secondly, it would mean that miners *must* track the state of *all* reflections in the simplex for some period of time so that they ensure the integrity of the reflection protocol. Although, given the Axiom of Availability, this is possible without significant overhead.

A practical method for treating proofs of reflection as witnesses that may be excluded/pruned is discussed in Section 4.3.

4.3 Segmented State

Traditionally, blockchain protocols have some *global* state and a state-transition function. For example, the Ethereum Yellow Paper says:

Quote

> Ethereum, taken as a whole, can be viewed as a transaction-based state machine: we begin with a genesis state and incrementally execute transactions to morph it into some current state. It is this current state which we accept as the canonical "version" of the world of Ethereum.
> [...]
> A valid state transition is one which comes about through a transaction. Formally:
>
> $$\sigma_{t+1} \equiv \Upsilon(\sigma_t, T)$$
>
> where Υ is the Ethereum state transition function.
>
> — *Dr. Gavin Wood; Ethereum Yellow Paper / Petersburg Version 41c1837, s2*

[47] Segregated Witness was a new "witness" structure introduced to bitcoin blocks, separate from the transaction merkle tree. The structure contains data required to check transaction validity but not required to determine the transaction effects. Introduced with BIP-141.

One of the reasons for this tradition is that transactions are (typically) permitted to depend on any part of the global state. For example: a Bitcoin transaction is permitted to spend any UTXO, and an Ethereum smart contract may interact with any other smart contract on the Ethereum blockchain.

However, it is not necessary for a protocol to allow *any and all* transactions to depend on global state. A protocol could specify that certain transactions may depend only on a strictly defined subset of global state, i.e., a well defined *segment* of global state that is independently calculable.

Simplex-chains can use this technique to their advantage by segmenting both transactions and state which are specific to Proof of Reflection. That way, the state of a simplex-chain's reflections can be calculated without needing to calculate the remaining state for that simplex-chain.

We could specify the state-transition of simplex-chains (using Ethereum's nomenclature) like this:

$$\begin{aligned}\sigma_{R,t+1} &\equiv \Upsilon_R(\sigma_{R,t}, T) \\ \sigma_{\star,t+1} &\equiv \Upsilon_\star(\sigma_{R,t} + \sigma_{\star,t}, T)\end{aligned} \tag{26}$$

Where, at some time t: $\sigma_{R,t}$ is the segment of state that is tracking reflections and headers; $\sigma_{\star,t}$ is the global state excluding $\sigma_{R,t}$: Υ_R is the state-transition function for the reflections segment; and $\Upsilon_\star$ is the state transition function for all remaining segments. Note that if T is a reflection-transaction (i.e., it contains headers to be reflected) then $\Upsilon_\star$ does nothing, and if T is any other type of transaction then Υ_R does nothing.

In essence Equation 26 shows that $\sigma_{R,t}$ depends *only* on the $\sigma_{R,t-1}$ state-segment and the current transaction, whereas $\sigma_{\star,t}$ depends on global state.

If simplex-chains are segmented in this manner, then miners will be able to calculate the reflection-state of other simplex-chains without calculating their complete state. This would allow them to deterministically calculate proofs of reflection for all other simplex-chains.

4.4 Exploiting Segmented State

Given that the reflection-segments of simplex-chains will contain mostly redundant data (i.e., headers), numerous optimizations are possible.

For example, it's not necessary for a miner's node to re-download reflected headers (which are part of other chains' blocks), since it can download them in advance and as they become available. We can reconstruct the PoRs root, provided that we know *which* headers are reflected and in what order. Transmitting the *hashes* of headers, only, reduces the effective size of simplex-blocks[48] from b to $\sim b \cdot (\frac{g+B_h}{2B_h})$, where g is the size of the relevant digest in bytes. For $g = 32; B_h = 112$, this reduces effective block size to $\sim 0.643b$ — an improvement of $\sim$35%.

However, *instead* of using that technique to *minimize bandwidth* we could instead use it to *maximize the number of simplex-chains.* If simplex-blocks dedicate $1/2$ of their capacity to reflections, and assuming +OP, then we can reduce that burden by $1 - 32/B_h \approx 70\%$, *or* we could increase the capacity for reflections by $B_h/32 \approx 300\%$!

Term

Header Omission (+HO): The UT protocol variant wherein miners/validators explicitly record *only* the hashes of reflected headers. A requirement is that block producers must eagerly download the headers of all simplex-chains and deterministically recalculate the relevant Proofs of Reflection.

We have just reached the +HO variant via *omitting proofs* (+OP). This time, however, it is not the *proofs* that are redundant, but the *headers.*

Does *header omission* with *explicit proofs* provide any advantages? Yes, in some cases.

[48]Assuming those blocks dedicate 50% capacity to transactions, and 50% to reflected headers (without PoRs).

Particularly, if miners include only the single missing merkle branch associated with each necessary PoR, then *no additional information* is required besides the *header* itself. Headers are trivial to acquire from the network, and each only needs to be acquired once, regardless of the number of PoRs it is a part of. Since the *hash of each header* is *part* of the missing PoR merkle branch, miners only need to provide *an ordered list of merkle branches* for full PoR verifiability. Additionally, these merkle branches *will be part of specific SPV proofs*, so that when a cross-chain SPV transaction (which uses those branches) is made, it can omit those parts of the proof (replacing them with a pointer).

This UT protocol variant is +HOPoRs: the combination of *header omission* (+HO) and *explicit proofs* (+PoRs). It may present decisive advantages for implementations of *simplex tilings* (which are introduced in Section 6).

4.4.1 Hash Compression & Truncation

Warning The following applies to PoW chains with compatible header PoW-hashing algorithms.

Consider a fairly normal PoW chain, in that the PoW algorithm compares the hashes of headers as a number with a target. That is, the output hash has a bunch of zero digits at one end (or can be losslessly converted into such a form). For simplicity, we'll assume that the zero digits are the most significant and are leading (and the least significant bits are trailing).

When we serialize this hash as a binary string, there is a trivial compression method. Since we *know* that one end of the hash has multiple zero bytes, we can replace this substring with the number of zero bytes replaced.[49]

This reduces the length of the hash (in bytes) from g to $g - z + 1$ (z being the number of zero bytes), *however*, the *security* of the hash is still $8g$ bits. This is a limited form of hash compression. In a mature network using +HO, reducing hash length by $\sim 1/4$ is possible, corresponding to an increase in maximum capacity of $\sim 1/3$ or so.

This kind of compression is possible (and valuable) because the hashes are *laden* with special properties by the PoW algorithm. There are two main properties we are concerned with. The first is that better header hashes, interpreted as numbers, are smaller. The second is subtle.

Consider a mature, healthy PoW chain, like Bitcoin. The proofs of work produced by such a chain are, through market feedback mechanisms, the *best* proofs of work that the market as a whole (i.e., the global economy) is capable of producing profitably. Therefore, the proofs of work are an approximate *measure* of the global capacity for hashing. More specifically, the number of leading zero-bits loosely encodes humanity's collective ability to produce arbitrary partial collisions via brute force. For example, Bitcoin block 879,273 has a difficulty around 1.1×10^{14}, corresponding to at least 78 leading zero-bits.[50] Given Bitcoin produces around 52600 blocks per year (and all else being equal), we can expect the best block hash produced in the last year to have around $78 + \log_2(52600) \approx 94$ leading zero-bits. Practically speaking, all the silicon in the world working for a year, singularly, on finding a partial SHA256 collision would not do much better than 94 bits. We can be confident in this *particular* prediction because the market for Bitcoin ASICs is mature – those ASICs use the latest fabrication processes, are numerous enough, and are orders of magnitude more efficient than general processors (CPUs, GPUs, etc). For less mature networks, or networks using algorithms that benefit less from ASICs, the difference will be greater and more dependent on external compute resources which are not reflected in the PoW difficulty. But, provided that $g \gg 2z$, this does not present an issue – the low value of z means we have more buffer until we pass the insecurity breakpoint, so these two forces roughly cancel out in all but extreme cases.

We will make use of this second property, that the leading zero-bits are related to the global hashing capacity, to justify the safety of *hash truncation* as an optimization.

[49] Since we're now dealing with a variable length byte-string, a typical encoding scheme would prefix this with the length of the bytes. From this we can recover the number of zero bytes, so we don't need to explicitly encode it.
[50] The block itself had 81 leading zero-bits.

Term

Hash Truncation (+T): The UT protocol variant wherein miners/validators refer to reflected headers using *only* the least significant half of the hash. This effectively halves the hash size in throughput calculations for +OP and +HO variants.

The idea behind +T is that the security of a truncated hash in bits, assuming $g > 2z$, is given by $8(g/2 + z)$[51], and that this is *always* sufficient, given that z is intimately related to the largest reasonable attack that we could expect at that moment. The $g > 2z$ condition is there for two reasons. First, if $2z \geq g$ then the security of the truncated hash is just $8g$ bits, like normal. Second, using significantly more than half the hash for PoW is problematic because the chance of collisions between valid headers increases dramatically. So, intuitively, there is some breakpoint that we cross as the bits used for PoW increases. Crossing that breakpoint indicates that the hash is no longer suitable for use in PoW – we've "maxed it out" and need a hash with more bits, or a hash that is harder to generate (or slower, etc). Therefore, there is also some similar and related breakpoint where the safety of hash truncation degrades. Perhaps it does not degrade completely, but at least enough that partial collisions (without the required PoW) of the least significant bits become practical to generate. Even though this shouldn't cause an issue for full and rigorous validation, it might open up DoS vectors, or other unforeseen exploits associated with optimizations or software patterns that might otherwise be safe. Treating that breakpoint as $g \approx 2z$ should provide us a reasonable safety margin to avoid such issues — if we get close to it, it's time to change the hash.

Combining +HO and +T gives us +HOT, the highest capacity variant of UT_1. In this configuration: headers are smaller, the information required for PoR regeneration is minimal, and the number of chains per simplex within given constraints is maximal. Since +T halves the effective hash length (which is all the data in the PoRs half of the block), the overall simplex capacity increases 2×.

$$\begin{tikzcd}
\boxed{\text{Conservative}} \arrow[d, dotted] & & & \\
\text{UT}_{\text{+PoRs}} \arrow[rr] \arrow[dr] \arrow[dd] & & \text{UT}_{\text{+OP}} \arrow[dr] \arrow[dd] & \\
& \text{UT}_{\text{+PoRTs}} \arrow[rr] \arrow[dd] & & \text{UT}_{\text{+OPT}} \arrow[dd] \\
\text{UT}_{\text{+HOPoRs}} \arrow[rr] \arrow[dr] & & \text{UT}_{\text{+HO}} \arrow[dr] & \\
& \text{UT}_{\text{+HOPoRTs}} \arrow[rr] & & \text{UT}_{\text{+HOT}} \\
& & & \boxed{\text{Maximal TPS}} \arrow[u, dotted]
\end{tikzcd}$$

Figure 16: Possible upgrade paths between UT variants, starting at $\text{UT}_{\text{+PoRs}}$ in the top left — the most conservative variant. Solid arrows show paths of increasing capacity.

4.5 Stateless Full Nodes and Fraud Proofs

A stateless full node (a.k.a. a stateless client) is a node that verifies all blocks, but does not actively track the current state of the chain. Instead, a stateless full node will use some minimal *witness data* (specific to each block) to verify that the state transition was valid. This is possible because *most* of the state tree remains untouched and thus the state root can be recalculated easily. The mutations to the state tree made by a block's transactions are all calculable from the witness data

[51] I use g here, even though it's measured in bytes, for consistency with the rest of the whitepaper. A more rigorous method would measure everything in bits instead, and avoid the somewhat awkward multiplication by 8, but it's not important for this discussion.

and the transactions, alone. In other words: the prior state root, the witness data, and the full block are all that is required to calculate the output state root of that block.

Aside

We won't focus too much on the specifics of stateless nodes here, since there are many existing resources[52] on the topic. Note that some stateless node designs require that each individual *transaction* include appropriate witnesses, rather than each block having a combined witness. Although we will design a method that supports transaction witnesses, we won't require that transactions include them.

Supporting stateless nodes introduces some constraints on the cryptographic design of state and blocks. Updates to the state tree must be computationally efficient — something like $O(\log m)$ complexity for m total state elements. Practically, it appears that this requires using an append only structure (e.g., a Merkle Mountain Range (Peter Todd; 2018)), or some kind of merklized prefix tree (e.g., a Merklix tree (deadalnix; 2016)). Additionally, headers should[53] commit to the state root, too, so that witnesses for individual transactions are minimal.

This is a good start, and we can now produce witnesses for each transaction.

However, the witness associated with a transaction *before* it is confirmed *will rarely* be the same as the witness *after* it is confirmed. The before-witness will prove some state against the prior block's state root. However, the state is almost always mutated before a transaction is executed — the state root is different. The after-witness should use, instead of the state root of the prior block, the root of the *intermediate* state immediately prior to the execution of that transaction. This way, the headers-only chain, the after-witness, and the transaction are the only data required to verify the validity of the transaction (assuming that the merkle proof of the transaction in the block also includes the intermediate state roots directly before and after the a transaction).

Why does this after-witness matter so much? Because it *is* the *fraud proof* for any fraudulent transaction.

Term

Fraud Proof: Cryptographic evidence that a transaction, block, or state transition was incorrect.

We do, however, need to ensure these mid-state roots are also recorded in the block. Typically, a block will commit to its transactions via something like the root of a merkle tree of those transactions. We can trivially modify this design by replacing each leaf with the corresponding {mid-state root, transaction} pair. Thus, if a block is invalid due to containing a fraudulent or otherwise invalid transaction, the fraud proof for that block consists of: the merkle branch proving the {mid-state root, transaction} pair, the pair itself, and the witness data for that transaction matching the mid-state root.

Additionally, such fraud proofs are all but *automatically* generated by full nodes during block verification — the witness data is exactly those state entries that the node requires to validate the transaction. In some cases, the witness data may not even be required if, for example, the transaction has an invalid signature. We could even optimize certain cases to reduce witness sizes by showing only the minimal data to prove that a particular condition was violated.

4.6 The PoR Graph

As a simplex ages, a complex graph of PoRs is produced (and it is a DAG). Each vertex represents a block in some simplex-chain, and the outgoing edges point to reflected blocks. An example section of a simulated PoR graph is shown in Figure 17.

[52]See: *Stateless Full Node* (Zack Hess), *The Stateless Client Concept* (V. Buterin), *Making UTXO Set Growth Irrelevant With Low-Latency Delayed TXO Commitments* (Peter Todd).

[53]This isn't *strictly* necessary — it could be recorded elsewhere in a block, e.g., the coinbase transaction. However, in that case, the effective header size becomes the actual header, plus the branch to the coinbase tx, plus the coinbase tx. Committing elsewhere in the block doesn't have any overhead for stateless full nodes (since they have the full block data, anyway), but it does increase witness sizes for individual transactions.

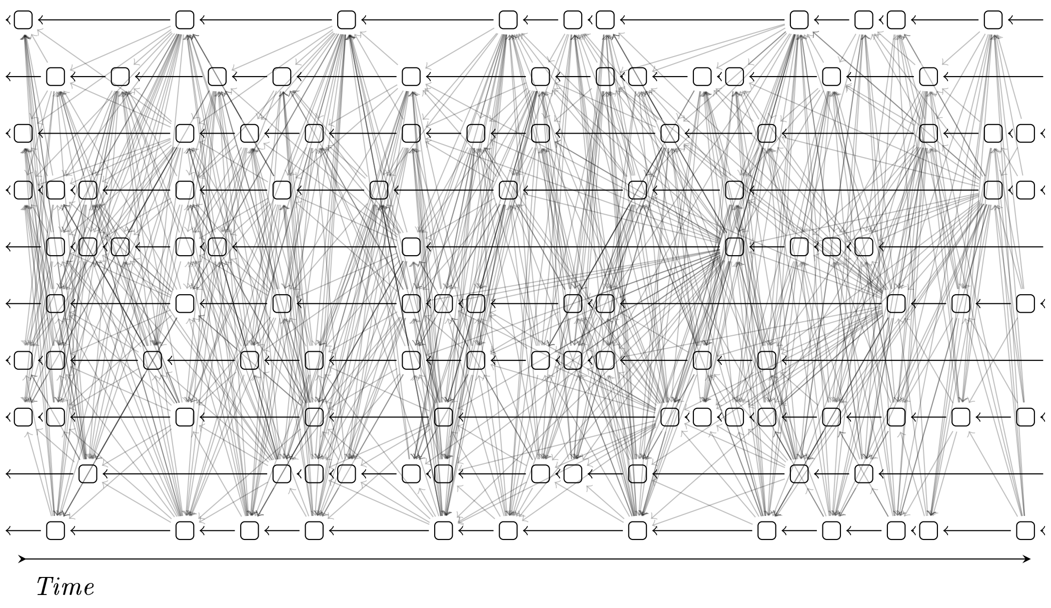


Figure 17: A simulated PoR graph of a 9-simplex over ∼10 block periods. Approximately 900 reflections are shown. Opaque horizontal arrows point from child to parent, and the semitransparent arrows point to reflected blocks.

Since there are N_1 chains, and each block has $\sim N_1$ reflections on average, the total number of reflections, per block period, is $\sim {N_1}^2 = O(c^2)$. Does this present a problem for fully validating nodes that wish to verify the entire PoR graph?

Recall that in Section 4.4, we took advantage of redundant data within the PoR graph. Particularly that, provided we knew *which* blocks were reflected and the *ordering* used to create the PoRs root, we could deterministically reconstruct those parts of any simplex-chain block. However, this relied on either tracking *the entire* PoR graph, or additionally downloading all missing merkle branches corresponding to PoRs used by the reflected block.

Why is so much information required? The reason is that each miner can choose *whether* to reflect any remote block (assuming that the block is available), and has freedom to choose the *position* of each reflected block in the merkle tree of PoRs.

Regarding the *position* of reflected blocks — this is easy to solve. All we need to do is ensure there is only one position that a block can be in. For example, we could require that reflected blocks are ordered based on the ordering of their genesis blocks (which is fixed). If two or more blocks from a single simplex-chain are reflected, we can use a sub-ordering method (of which there are many options). This gives us a total ordering, meaning that we only need to know *which* blocks were reflected to reconstruct this segment of a block. Alternatively, we could use a merklized prefix tree which achieves the same goal. Either way, we have reduced the required information from a *list* of block hashes to a *set* of them.

It is clear that miners need some freedom to choose *whether* to reflect a block or not. For one, miners must be able to choose to reflect *new* blocks as they appear. Additionally, due to the Axiom of Availability, miners must not reflect unavailable blocks, even if a valid PoR can be generated. However, because all honest miners obey the Axiom of Availability, there is never a reason for an honest miner *not* to reflect a block that is available — doing so will always increase the overall chain-weight of that miner's draft block.

So, if an honest miner's block, L_i, reflects some block, R_k, then any block reflected by R_k must also be available (since the Axiom of Availability is recursive). If a block is available then a PoR is possible. Therefore, an honest miner should reflect all non-local blocks that were reflected by R_k but have not already been reflected by an ancestor of L_i.

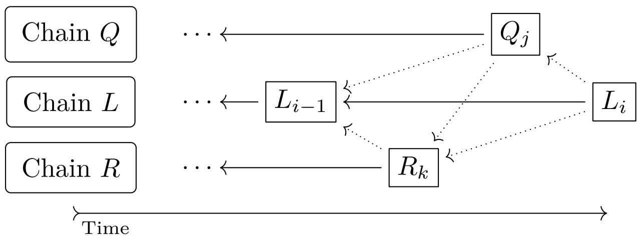
Let's call this a state of *maximal reflection*, and say that such a PoR graph is *maximally reflective*.

4.6.1 Maximally Reflective PoR Graphs

Specifically, we can say this about any PoR graph in a state of maximal reflection: for every block L_i, and for every block R_k reflected by L_i, then every block b that is reflected by R_k satisfies one of: b is reflected by L_i; or b was reflected by an ancestor of L_i; or b is an ancestor of L_i.

Notice that, if L_i reflected any blocks, there will always be at least one block that *was* reflected by L_i that was *not* reflected by any other block that was reflected by L_i. Let us call these blocks the *tips* of the PoR graph from L_i's perspective. PoR graph tips might reflect other blocks reflected by L_i, too.

For example, consider this PoR graph segment between 3 chains L, Q, and R:



Here, L_i reflects both Q_j and R_k, and Q_j also reflects R_k. Thus, from L_i's perspective, Q_j is a *graph tip*; and, from Q_j's perspective, R_k is a *graph tip*.

Starting from L_{i-1}, we could completely describe this section like so:

- R_k reflects L_{i-1}; and
- Q_j reflects $\{R_k, L_{i-1}\}$; and
- L_i reflects $\{Q_j, R_k\}$.

If the PoR graph is not maximally reflective, then we must specify the complete set of reflections for each block. However, *when maximally reflective*, most of that information is redundant (about half in this example).

We *do* still need some information for each block, however, we can recreate the full PoR segment with less information — some of the reflections are *implied* and can be omitted:

- the PoR graph segment is maximally reflective; and
- R_k reflects the tip L_{i-1}; and
- Q_j reflects the tip R_k; and
- L_i reflects the tip Q_j.

We can see that the maximally reflective quality has at least two important properties. Firstly: we have *drastically*[54] reduced the amount of information needed to completely describe the graph segment: $O(c^2) \to \sim O(c)$ — an example is shown in Figure 18. Secondly: the PoR graph, according to each chain, *converges* into a single, consistent PoR graph. In essence, any block in a maximally reflective PoR graph shares a *common history* with all prior blocks in the graph.

Is there an intuitive explanation for the reduction in complexity of the PoR graph? Yes. To start with, let us assume that there is no *propagation delay* — i.e., all miners instantly receive, validate, and incorporate all blocks into the PoR graphs of their draft blocks. In this case, no blocks are ever mined simultaneously, and so each block will reflect exactly one PoR graph tip. Since there are $O(c)$ many chains, and we need to add $O(1)$ extra information per header (the reflected tip), this is $O(c)$ extra information all up.

In practice, blocks *will* sometimes be mined simultaneously. So, even if *most* of the time a block only reflects one tip, *some* of the time that block will reflect multiple tips. The frequency of simultaneous blocks is dependent on the number of chains (N_1), the block frequency of each chain

[54]The theoretical overhead is still $O(c)$ per block, but practically we expect the overhead to be a similar size to the header which is $O(1)$. It's explained shortly.

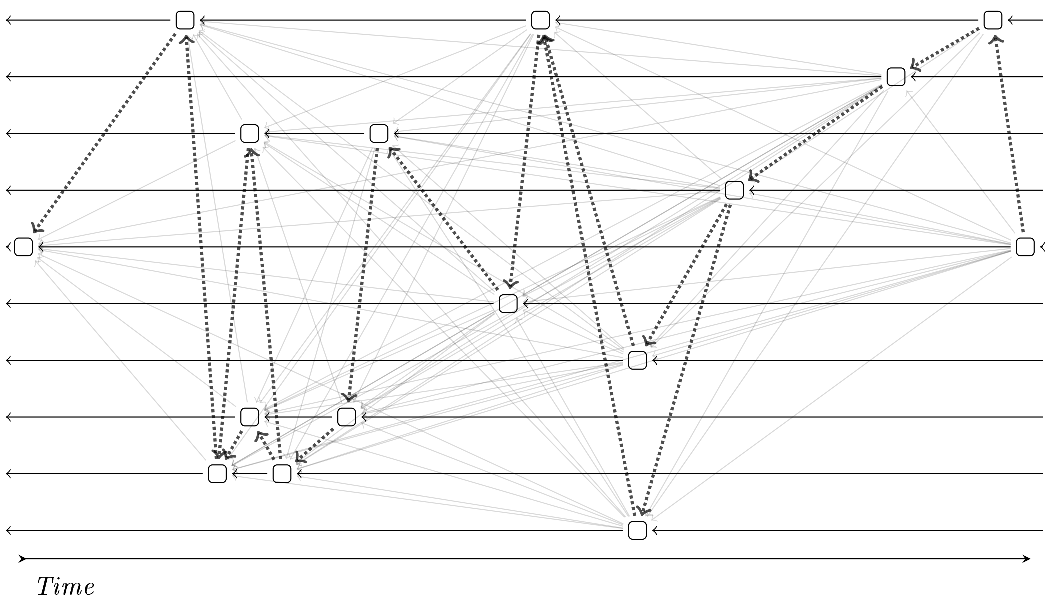


Figure 18: This demonstrates how maximally reflective PoR graph segments with $O(c)$ many chains can be described using $\sim O(c)$ information. Thick dotted arrows represent reflections of PoR graph *tips*, and semitransparent arrows show other reflections — these reflections and implied via the reflection of the graph tips. This also shows that the DAG of tip-reflections is the *longest* path of reflections between two blocks.

(B_f), and some average propagation[55] delay (ϕ seconds). Two blocks are simultaneous if they are mined within $\sim\phi$ seconds of each other. In the ϕ seconds before that most recent tip was mined, we expect that $\sim N_1 B_f \phi$ other tips were mined (from any simplex-chain). Unless a block contains no reflections (which happens with probability $1 - {N_1}^{-1}$), it must reflect at least one graph tip. So, we expect each block to reflect $1 + N_1 B_f \phi$ tips. For $N_1 = 100, B_f = 1/15, \phi = 1$, we expect each block to be simultaneous with $\sim$7 others, and each block to reflect 8 graph tips on average.

Therefore, the overhead to reconstruct the PoR graph — and every PoR segment of every simplex-chain block — is, on average, $g(1 + N_1 B_f \phi)$ bytes per block, where g is the length of a hash. As g, 1, and B_f are constants, and if we treat ϕ as constant, the overhead thus has complexity $O(c)$ *per simplex-chain*. Since we're tracking $O(c)$ chains, the overall load is $O(c^2)$.

The total bandwidth required to track the PoR graph (excluding verifying the availability of blocks) is: the overhead per block, plus the block header, all multiplied by the network-wide blocks per second rate. Thus, the total bandwidth is $N_1 B_f (B_h + g(1 + N_1 B_f \phi))$ B/s — which is $O(c^2)$.

However, we estimated before that we should expect to require 7 graph tips on average per block (which seems much less than $O(c)$). Practically speaking, the overhead is only significant when the header size is much smaller than the overhead: $B_h \ll g(1 + N_1 B_f \phi)$. If we plug in our rough numbers from before, we find $g(1 + N_1 B_f \phi) \approx 250 \approx 2B_h = O(1)$.

There *is* a practical limit (depending on $B_f \phi$) for how large N_1 can grow before this overhead is non-negligible. *However*, if we capped the expected overhead to, say, 256 bytes — $g(1 + N_1 B_f \phi) \leq 256$ — then we find $N_1 \leq 7(B_f \phi)^{-1} \approx 105 \cdot \phi^{-1}$. Capping the overhead to 128 bytes yields $N_1 \lesssim 45 \cdot \phi^{-1}$. Note that maximal values of N_1 are inversely proportional to B_f, so modifying B_f doesn't change the expected overhead.

4.6.2 Optimization or Contradiction?

We calculated the overhead to be $g(1 + N_1 B_f \phi) = O(c)$ — this is because we assumed that $O(B_f \phi) = O(1)$. However, if this were really the case, we should be able to optimize the PoR

[55]In particular, we are concerned with propagation *between miners*, not the network as a whole.

sections of blocks (similar to Section 4.4).

We could try to design the protocol like this: *instead of recording all reflections, each block records only the minimum necessary to reconstruct the PoR graph.* This would increase the overall bandwidth required to track the PoR graph, but our expression remains the same: $N_1 B_f(B_h + g(1 + N_1 B_f \phi))$.

Since the overhead is $O(c)$, and recording all reflections is also $O(c)$, can we swap in this new process to increase scalability?

The old PoR sub-block takes up $N_1 B_f \alpha$ B/s of capacity (where α represents the bytes per reflected simplex-chain and $\alpha \in [16, 512]$).

The new PoR sub-block instead takes up $B_f \cdot g(1 + N_1 B_f \phi)$ B/s of capacity — we exclude the block's header since that's already part of the block. If $k_{1,B}$ B/s of a chain's capacity is dedicated to reflections, then we have:

$$\begin{aligned} k_{1,B} &= B_f g(1 + N_1 B_f \phi) \\ N_1 &= \frac{1}{B_f \phi}\Big(\frac{k_{1,B}}{B_f g} - 1\Big) \end{aligned} \tag{27}$$

Setting $k_{1,B} = 1{,}500$ B/s and evaluating yields $N_1 \approx 10{,}500 \cdot \phi^{-1}$.

Of course, ten thousand chains is a lot, and full tracking the PoR graph would require ~15 MB/s of bandwidth and around 500 TB of storage per year. Each block would reflect, on average, 10k other blocks. Every block period, keeping up with the PoR graph would require calculating 10k PoRs roots — although there are significant optimizations to avoid recalculating these from scratch.

Aside I don't know about you, but I'm getting the suspicion that this isn't really an $O(c)$ load.

Since this conflicts with our notion of an $O(c)$ load, let's examine one of our premises: that $O(N_1 B_f \phi) = O(c)$. Certainly $O(N_1) = O(c)$, and $O(B_f) = O(1)$, but does $O(\phi) = O(1)$?

There is a good reason to think $O(\phi) = O(1)$. At some point, propagation delay is limited by the speed of light. So latency must have a definite minimum, which is a constant.

However, the physical limits on latency do not dominate real-world latency yet. Near-future networking technologies, like LEO satellite constellations, will reduce latencies over significant distances. Moreover, even as the best-case minimum latency approaches physical limits, technology will continue to improve latency in *bandwidth intensive* contexts.[56] Therefore, $O(\phi)$ *should not* be considered $O(1)$, but rather should be considered $O(c^{-1})$: as technology improves, ϕ decreases.

If $O(\phi) = O(c^{-1})$, this resolves the contradiction. It means that $O(N_1 B_f \phi) = O(1)$. That matches our intuition from earlier, *and* means that maximally reflective PoR graphs really are only $O(c)$ complex, rather than $O(c^2)$.

We can also explain why we had such a large N_1 estimate before — we took an $O(1)$ load and blew it up to an $O(c)$ load. So we effectively changed $O(N_1)$ from $O(c) \to O(c^2)$. That meant that network-wide bandwidth and storage requirements went from $O(c^2) \to O(c^3)$.

There is also an argument that ϕ depends on the number of nodes in the P2P network — that would take the form of a $\log_b(n)$ term in $O(\phi)$. While this is true for naive gossip protocols, in practice we see that miners will go to some effort to specifically reduce latency for block propagation. Under near-ideal conditions, miners can propagate new blocks and reflections to one another with near-zero local overhead (compared to the milliseconds of latency between them). Since this propagation is constrained by the latency between these special nodes, and not by the size of the network as a whole, we will ignore any $\log_b(n)$ component of ϕ.

[56]This is something we didn't consider when estimating N_1 in Equation 27 — that the propagation delay will increase as the excess capacity of nodes' available bandwidth decreases. Like all such systems, response time is inversely proportional to excess capacity.

4.6.3 Axiom of Maximal Reflection

It should be clear that there are decisive advantages for UT if the PoR graph is maximally reflective. Particularly, we can accurately describe the PoR graph entirely via *new reflections*, greatly reducing PoR overhead. Thus, we will require it via a protocol rule: the Axiom of Maximal Reflection.

Axiom

Axiom of Maximal Reflection

Principle: A miner must reflect all possible blocks.

Predicate: A block, L_i, is invalid unless, for each reflected block R_k:

i. R_k is valid under this Axiom; and
ii. each block that was reflected by R_k or is a parent of R_k:
    a. is reflected by L_i; or
    b. is an ancestor of L_i; or
    c. was reflected by an ancestor of L_i.

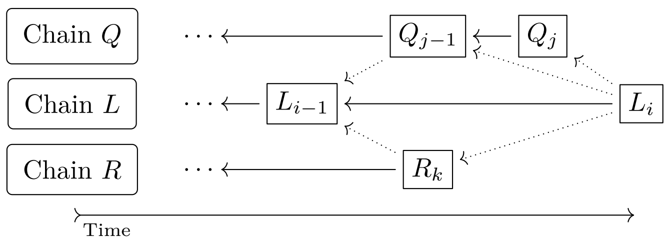
It is worth noting that this axiom does not impact honest miners — it is in their interest to produce maximally reflective PoR graphs. However, this *does* impact dishonest miners since reflecting other chains becomes all-or-nothing.

If a dishonest miner is determined to selectively reflect only *some* simplex-chains, their valid blocks cannot reflect any other blocks that are less selective. Honest miners, on the other hand, *can* reflect the dishonest miner's blocks. Thus, the Axiom of Maximal Reflection creates an asymmetry between honest and dishonest miners.

4.6.4 The Longest PoR Chain

Figure 18 shows us an interesting feature of maximally reflective PoR graphs: between two given blocks, the union of the paths via PoR graph tips both *visit every block* and are the *longest* paths between any two blocks. This is somewhat self-evident: since the PoR graph is a DAG, there is no way to return to a block once we have visited it. Thus, if a path exists that visits every block, then it must be a longest path because there are no unvisited blocks. A longer path would need to visit more blocks, but there are no more blocks, therefore no longer path exists.

Although it makes sense to refer to these paths as the *longest* — this isn't always strictly true. First, we should notice that multiple paths are possible, and are expected to occur when blocks are created near-simultaneously. In some cases, these paths can be different lengths. For example:



Here, we can see that L_i reflects the tips $\{Q_j, R_k\}$. There are two paths from L_i to L_{i-1} in this case: one via $Q_j \to Q_{j-1}$, and the other via R_k. Both are a longest path in the sense that they are locally maximal: no paths including R_k (or Q_j) are part of a longer path to L_{i-1}.

So, really, what we have identified is the DAG of the union of the longest paths between two blocks. This is exactly the information that is necessary to reconstruct a maximally reflective PoR graph.

Let's refer to this as the *longest PoR chain*, even though it is technically a DAG. Of all possible PoR DAGs that both include all edges leading to a graph tip and visit every block, the *longest PoR chain* has the fewest edges, and is thus the most chain-like of those DAGs. Additionally, in Section 4.8.7 we will discuss some symmetries between blockchains and the longest PoR chain.

Intermission and Note

Aside

Let us return to the Axiom of Maximal Reflection again for a moment. I would have said something earlier, but I wanted to give you some time to chew, first. If you were left wondering *What happens when R_k reflects an L block that is not an ancestor of L_i?* then I say *'well spotted'* to you — finding criticisms on a first reading is worthy of praise.

Now, the answer to the question: on the one hand, if the L chain deliberately forks, and two distinct chains emerge, then they should reflect one another like any other pair of simplex-chains. On the other hand... well the other hand is a problem that we can't solve, yet — we'll ignore it for now and return to it in Section 4.8.7.

4.6.5 Generalizing NIPoPoWs

Warning

This section is speculative; it's about how NIPoPoWs might be possible for the longest PoR chain. Subsequent sections regarding NIPoPoWs will, likewise, be speculative. UT won't depend on NIPoPoWs, but they are useful and it would be good to support them, if possible.

4.6.5.1 NIPoPoWs Primer

Prior work: Non-Interactive Proofs of Proof-of-Work (Kiayias, Miller, and Zindros; 2018).

Quote

Non-Interactive Proofs of Proof-of-Work (NIPoPoWs) are short stand-alone strings that a computer program can inspect to verify that an event happened on a proof-of-work-based blockchain without connecting to the blockchain network and without downloading all block headers. For example, these proofs can illustrate that a cryptocurrency payment was made.

— *What are NIPoPoWs? (nipopows.com; 2019)*

NIPoPoWs are possible due to the statistical properties of proofs of work. They require an additional interlink structure between blocks. The interlink structure is a kind of skip-list, where a block at height h links back to $\sim\log_2 h$ other blocks.

The insight that enables NIPoPoWs (and PoPoWs more generally) is that some PoWs will be *much* lower than the difficulty target, and that the frequency of these special PoWs is dependent on the total number of PoWs that have been produced. That is: if the PoW target is 2^{184}, then all valid PoWs will be $\leq 2^{184}$, 50% will be $\leq 2^{183}$, 25% will be $\leq 2^{182}$, and so on. We can generalize this via the statement: $P(\text{PoW} \leq 2^{184-\mu}) = 2^{-\mu}$ — the probability that the proof of work is less than or equal to $2^{184-\mu}$ equals $1/2^{\mu}$. In essence, μ is a measure of the *excess capacity* of a block's PoW: how many orders-of-magnitude lower *could* the target have been, given the *actual* PoW.

When a block has $\mu > 0$, it is a μ-superblock, and it is these superblocks that the interlink structure uses. Particularly, for each integer value of μ, each block links back to the most recent μ-superblock. The chain of μ level blocks is the μ-superchain. Note that the 0-superchain (or 0-chain) is identical to the blockchain itself. This interlink structure is partially shown in Figure 19, which shows each μ-superchain according to the perspective of the right-most block. The full interlink structure is shown in Figure 20.

Intuitively, a NIPoPoW works because it uses the (very short) top-most chain of μ-superblocks as a basis to hone in on a target block (denoted $\mathcal{C}[-k]$ in the NIPoPoW paper) a few confirmations behind the chain tip. This is done by progressively inspecting the most recent segments of lesser and lesser μ-superchains to establish the expected chain-weight that has been contributed to that target block.[58] This is the main *suffix proof*, as it proves the blocks at the recent end (suffix)

[57]Source diagram: https://nipopows.com/images/hierarchical-ledger.png

[58]The NIPoPoW algorithms don't do this exactly, but they embody similar and convergent knowledge.

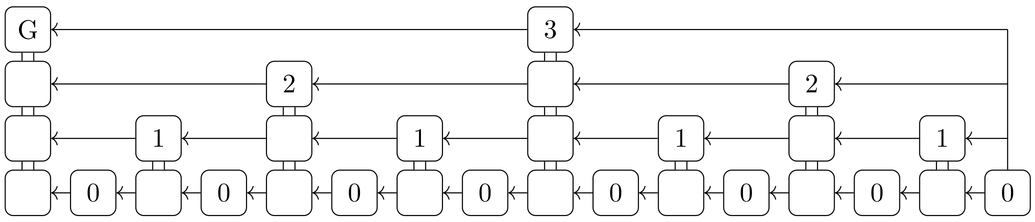


Figure 19: The μ-superchains used by NIPoPoWs. Each column represents one block. The y-axis indicates the μ level of each block. A μ-superblock is also a $(\mu - 1)$-superblock. *Credit:* the structure and content of this figure is credited to the authors of the NIPoPoW paper (Kiayias, Miller, Zindros) — this figure is a recreation of one of their diagrams.[57]

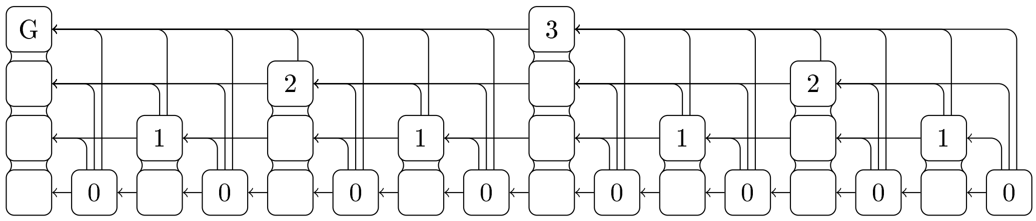


Figure 20: The full interlink structure used by NIPoPoWs for the chain segment shown in Figure 19.

of the blockchain. (Additionally, the most recent k blocks ($\mathcal{C}[-k :]$) are included alongside the proof.) The work required to generate the main suffix proof is (approximately) as much work as has been contributed to the entire blockchain history. Thus, an adversarial NIPoPoW requires (approximately) as much work as the total chain-weight of the honest chain, so NIPoPoWs are secure.[59]

Now that the chain-head and relevant μ-superchains have been proven, arbitrary blocks in the chain's history can be proven. This is done by 'following down' from a top-level μ-superblock to the target block, following the interlink structure. Starting with the *earliest* μ-superblock between the target block and $\mathcal{C}[-k]$, up to μ levels are progressively descended until the target block is reached. 'Following down' means: for a μ-superblock, check if the previous block at level $(\mu - 1)$ occurred *after* the target block — if so, that block becomes the next in the series, otherwise decrease μ by one and repeat with the current block. The proof terminates at the target block. The blocks that are traversed comprise the *infix proof* for that block.

An example of a NIPoPoW with a suffix and infix proof is shown in Figure 21.

Notice that there is no concept of *state* used when reasoning about NIPoPoWs — we're only considering a chain of interlinked blocks with PoWs. Put another way: there is no requirement that these blocks belong to the same blockchain; only the PoW and interlink structure matters.

So, at first glance, it appears that the longest PoR chain might work with NIPoPoWs, or some modified variant, at least.

There are two important caveats that are mentioned in the NIPoPoW paper; these are problems for us and we will have to address them.

First: the authors assume a single chain of blocks, not a graph. Thus, *any* simultaneous blocks in the PoR graph are an issue.

Second: the authors assume a constant difficulty. For a traditional blockchain, this isn't too much of a problem. For UT, however, we ideally want uncoupled simplex-chain difficulties, even if they use the same hashing algorithm. Speaking of which, how does a NIPoPoW work if more than one hashing algorithm is used?

[59]This is a simplification — see Non-Interactive Proofs of Proof-of-Work (Kiayias, Miller, and Zindros; 2018) for full details.

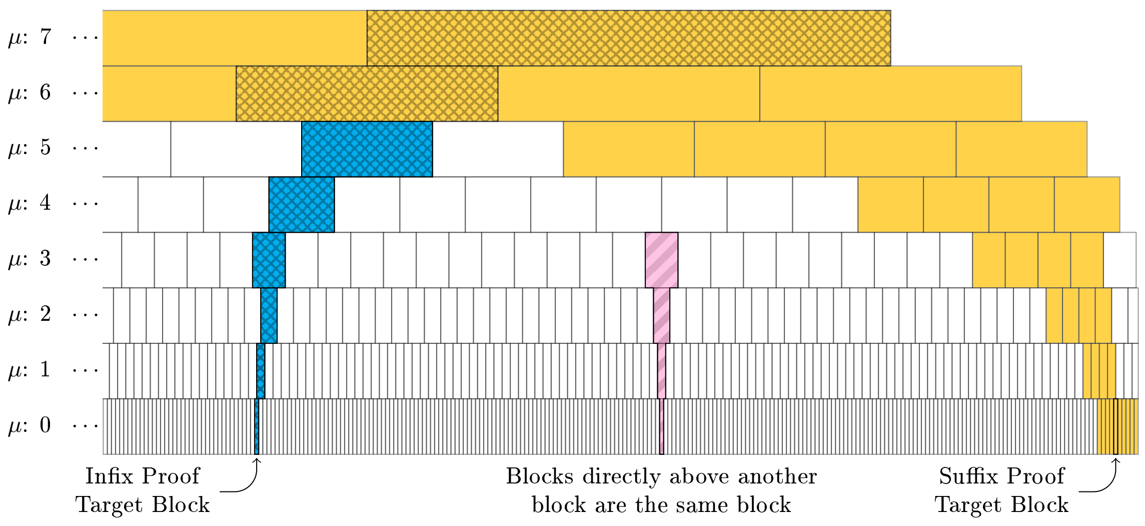


Figure 21: The blocks of a traditional chain that comprise a NIPoPoW. For a given $\mu > 0$, each superblock at that level is *the same* block as the one directly below it; only half of the blocks at level $\mu - 1$ are also μ-superblocks. Yellow blocks are part of the main suffix proof. Patterned blocks comprise the infix proof of the target block. Patterned blue blocks are unique to the infix proof. The striped pink and gray block is an explanatory aid and is a 3-superblock.

4.6.5.2 Non-Interactive Proofs of Proof of Work Reflection (NIPoPoWRs)

Before we discuss the problems mentioned above, it is important to note that the method we will devise cannot make use of converted work from non-PoW blocks (e.g., if some simplex-chain used PoS, or PoA, etc). That's because there is no real sense to such a block being 'lucky' in the way that PoW superblocks are. We can still link back to non-PoW blocks as 0-blocks, though (i.e., they're at the lowest μ level).[60]

Thus, these will not be NIPoPoRs — this method requires PoW, so, in terms of PoRs, we are only concerned with PoW reflection, rather than all reflection.

Aside

You should know, I am only confident about the *principles* of NIPoPoWs, not the specifics. So, I claim the following only as an *in principle* argument. Further diligence is required.

Problem: DAG Compatibility If the longest PoR chain is a DAG, then the interlink will need to be a DAG, too. Sometimes, two (or more) blocks will be simultaneous, so we will have two (or more) graph tips in those cases. When that happens, we will always have at least two 0-superblocks (since a 3-superblock is also a 2-superblock, etc). In fact, at and below the μ level of the lightest graph tip, all interlink levels will point back to all graph tips.

We are therefore dealing with μ-superDAGs, not μ-superchains. The existing NIPoPoW `followDown`[61] algorithm should work fine for high level μ-superblocks. Since simultaneous blocks at such levels will be exceptionally rare, we can expect higher μ-superDAGs to be much more chain-like than the longest PoR chain.

At lower levels (or any level), it doesn't matter which path to the target block we take (if more than one is even available). The PoR graph has no conflicts and no concept of a "doublespend" — rather, it is a massive graph of interconnected merklized tree structures where vertexes are merkle

[60]Note that, currently, this implies that non-PoW blocks are considered to contribute some work, which they do not in this context (it is not verifiable). We will resolve this conflict shortly — at the end of the section, it will make sense to consider them 0-blocks at the lowest μ level.

[61]The `followDown` algorithm produces the necessary blocks to connect a superblock to a proceeding regular block. It is defined in Section 5.1 and Algorithm 6 of Non-Interactive Proofs of Proof-of-Work (Kiayias, Miller, and Zindros; 2018)

roots and comprise the leaves of future vertex-roots; thus, edges are merkle branches. Put another way, there's no sense of *priority* between siblings of the same μ level. This is not true for chains using GHOST (or block-DAGs)[62] — in those cases it is possible for an uncle block to be of a higher μ level than its non-uncle sibling, so this construction doesn't cover those cases. But, in the case of the longest PoR chain, we are *only* concerned about *whether* a block exists — it doesn't matter *how* we get there, just that we *can* get there.

Curiously, this change is an alternate solution to the problem of an attacker "[breaking] chain superquality with non-negligible probability" addressed in the NIPoPoW paper (Section 7).

The attack is one that allows an attacker to manipulate the public chain's "superquality" (a measure of the frequency of μ-superblocks), such that fewer superblocks become canonical. In essence, the attacker selectively performs a selfish-mining-like attack whenever a significant μ-superblock is found. Their goal is to mine an alternate longest chain, such that the superblock becomes stale. This then allows the attacker to produce, in private, a superior NIPoPoW (compared to that of the public chain) for fewer resources than should be required.

The authors solve this problem via introducing a concept of "*goodness*" — which is basically that proofs must be succinct and "*good at every level*" (when this is not possible, provide the whole chain instead). The authors then show that, when goodness is not violated, the attack is impractical.

In our modified graph-based version, though, there is no such problem: blocks cannot become stale. Additionally, the Axiom of Maximal Reflection guarantees that the interlink history will fully span the entire PoR graph.

Problem: Variable Difficulty and Multiple Hashing Algorithms There are really two problems here for us to think about. If we can solve the problem of blocks using different hashing algorithms, then we will need to handle variable difficulty anyway, so let's work on the critical path first (variable difficulty).

Let's consider the superchains of two very similar traditional chains, $\mathcal{A}$ and $\mathcal{B}$, using the same PoW algorithm. Since these chains use the same hash, we can scale their respective μ values depending on their relative difficulties.

If the difficulty of $\mathcal{B}$ is half that of $\mathcal{A}$, then we can convert $\mu_\mathcal{A}$ levels into $\mu_\mathcal{B}$ levels by adding 1. In this way, all blocks in $\mathcal{A}$ are at level $\mu_\mathcal{A} = 0$ which is equivalent to level $\mu_\mathcal{B} = 1$.

We can observe this equivalence via the following thought experiment: what does $\mathcal{B}$'s 1-superchain look like if $\mathcal{B}$'s block frequency is twice that of $\mathcal{A}$? First, we should expect that, for a given period, there are twice as many $\mathcal{B}$ blocks as there are $\mathcal{A}$ blocks. Since about half of $\mathcal{B}$ blocks will have $\mu_\mathcal{B} \geq 1$, there will be approximately as many $\mathcal{B}$ blocks in the $(\mu_\mathcal{B} = 1)$-superchain as in $\mathcal{A}$'s $(\mu_\mathcal{A} = 0)$-superchain.

This makes sense: there are twice as many blocks at half the difficulty, so we expect the same hash-rate to be mining on each chain. If both chains have the same hash-rate, then they should produce the same number of blocks at a given $\mu_\mathcal{A}$ level. Since $\mathcal{B}$ has the lower difficulty, this is not true for a given $\mu_\mathcal{B}$ level — $\mathcal{A}$ produces *no* blocks at level $\mu_\mathcal{B} = 0$.

Now, let us modify $\mathcal{B}$'s protocol in a strange way. We'll say that $\mathcal{B}$ blocks *must* explicitly include all transactions (in sequence) from all ancestor blocks between the block itself and the prior $(\mu_\mathcal{B} \geq 1)$-superblock. We'll also assume that there are suitable optimizations such that this isn't an issue for the $\mathcal{B}$ network. This modification means that no $\mathcal{B}$ NIPoPoW ever needs to descend to below $\mu_\mathcal{B} = 1$ again.

For all intents and purposes, $\mathcal{A}$ and $\mathcal{B}$ produce almost indistinguishable NIPoPoWs (excepting, of course, that they prove blocks in different chains). Without analysis of the $\mathcal{B}$ protocol, and ignoring that all $\mathcal{B}$ proofs end at level $\mu_\mathcal{B} = 1$, there is no sign that the $\mathcal{B}$ chain has twice as many blocks at half the difficulty. The proofs are similar sizes, descend the same number of μ levels, have similar

[62] GHOST and block-DAGs are the specific focus of Section 4.8, and we'll discuss this issue properly, then.

numbers of blocks at each (adjusted) μ level, etc. So we might ask the question: how do we know that $\mathcal{A}$ doesn't produce blocks at level $\mu_{\mathcal{A}} = -1$, similar to $\mathcal{B}$?

The point of this is to demonstrate that μ levels are relative and convertible between $\mathcal{A}$ and $\mathcal{B}$, and there is no appreciable difference based on any μ-offset.

Therefore, we can express $\mu_{\mathcal{A}}$ and $\mu_{\mathcal{B}}$ in objective terms by measuring them relative to an arbitrarily low difficulty (e.g., 1). Let's call the μ level in terms of a difficulty of 1 the *objective* μ level.[63] A consequence of this is that duplicate interlink entries (over sequential levels) will be common when there is a large difference between difficulties (i.e., $\mu_{\mathcal{A}} \gg \mu_{\mathcal{B}}$). Thus, we should also modify the interlink structure to prepend the μ level to each group of previous blocks at that level. This allows us to omit the duplicates, assuming we use an appropriate cryptographic structure (e.g., a merklix tree seems suitable at first glance).

This works because no proof for either $\mathcal{A}$ or $\mathcal{B}$ will ever need to descend below the minimum objective μ of that chain's difficulty, and the statistical properties of all higher μ levels are maintained. It is no easier for an attacker to generate a malicious NIPoPoW than it would be otherwise.

Further, this is consistent with the observation that, if $\mathcal{A}$'s hash-rate doubled, then it would appear as though the minimum μ level increased by 1 (assuming that μ levels are not recalibrated according to the current difficulty).

However, this method is only partially consistent with the notion of *goodness* introduced in the NIPoPoW paper — the statistical distributions of μ-superblocks will only hold until the objective μ level of a chain's difficulty. We will ignore this problem as the graph-interlink structure has no need of a goodness measurement.

This resolves the first part of the problem: variable difficulties.

With regards to multiple hashing algorithms, PoR itself has already laid the groundwork for a solution: the conversion of work. Recall that in Equation 2 we derived $\text{ConvWork}_{R \to L}$: a function for converting R-hashes $\to$ L-hashes.

We can use this to calculate a PoW's equivalent μ level in terms of another hash. We can think of converting a block's weight as starting with the chain R target, converting that to the difficulty (R_d hashes/block), applying $\text{ConvWork}_{R \to L}(R_d)$, and converting this to the L target. Similarly, we can convert μ levels using the same process, except that we replace the *actual* target with the *lowest possible satisfiable target* for the block in question.

However, these μ levels are only comparable after linear scaling, so we need to differentiate them. From here out, we only need objective μ values, so let's use μ_R to refer to the objective μ of the R chain, and μ_L to refer to that of the L chain. We should note that, for a chain C, if we want the μ_C level relative to the difficulty, we just need to subtract $\log_2(C_d)$ (which is the minimum objective μ_C level of any chain C block).

Conversion Example: let's start with an R chain PoW target of 2^{184} — corresponding to a difficulty of 2^{72}. Next, a block is produced with a PoW of 2^{181}; it has an objective μ_R of 75. The minimum target that the block would satisfy is 2^{181}, which corresponds to a maximum difficulty of 2^{75} (R-hashes). Note that $\log_2(\text{max. difficulty}) = \mu_R$. We then convert this via $\text{ConvWork}_{R \to L}(2^{75})$ to get L-hashes. All that remains is to take and take $\log_2$ to get the converted objective μ_L.

What can we say about μ_R's relationship with μ_L? Consider that, in the SRT context, Equation 1 simplifies to $L_d R_r (R_d L_r)^{-1}$. Therefore:

$$\begin{aligned}
2^{\mu_R} \cdot L_d R_r (R_d L_r)^{-1} &= 2^{\mu_L} \\
2^{\mu_R} \cdot R_r / R_d &= 2^{\mu_L} \cdot L_r / L_d \\
\log_2(2^{\mu_R} \cdot R_r / R_d) &= \log_2(2^{\mu_L} \cdot L_r / L_d) \\
\mu_R - \log_2(R_d) + \log_2(R_r) &= \mu_L - \log_2(L_d) + \log_2(L_r) \qquad (28) \\
\mu_R + \log_2(L_d R_r) - \log_2(L_r R_d) &= \mu_L \qquad (29)
\end{aligned}$$

[63] The difficulty is the expected number of hashes to find a solution, so a difficulty of 0 (or below) is impossible.

Note that $\mu_R - \log_2(R_d)$ (Equation 28) is the block's relative μ_R level (and similarly for μ_L on the right-hand side).

All that remains is for each block in the longest PoR chain to calculate objective μ values of the same type using Equation 29.

Earlier, we determined that objective μ values of the same type can accommodate variable difficulties. Thus, we have resolved both parts of the problem.

NIPoPoWRs: Summary We have sketched the explanatory framework and construction of a variant of NIPoPoWs that is compatible with the longest PoR chain: NIPoPoWRs.

NIPoPoWs were modified in the following ways:

1. Convert (via PoR) each chain's native μ values into a common type for the interlink structure.
2. Use objective μ values to accommodate variable difficulties.
3. Replace the tree-interlink structure with a graph-interlink structure.

The Axiom of Availability means that the PoR graph does consensus on which blocks exist. The Axiom of Maximal Reflection means that the interlink structure spans the entire PoR graph. Thus, any path to a block is a valid path and we don't need to introduce the notion of *goodness*.

We should also note that this construction does not (yet) work for block-DAGs (including GHOST). We will continue this construction in Section 4.8.8.

Aside

Although we are considering a graph-interlink structure, we should note an obvious optimization. First, let's notice that: when introducing multiple back-links to blocks of equal μ level, those new blocks *must* be a subset of the PoR tips recorded in the header (since there might be other tips that are of a lesser μ level). Thus, we don't actually need to include both/all in the interlink structure, since we always have access to both/all paths via the PoR tips in the header, which is already part of the NIPoPoWR.

4.7 Confirmation Times

A confirmation is a *discrete* event that occurs when a block is produced. When an attacker is performing a hash-rate based doublespend attack, they are, effectively, racing the honest network; that race is measured in confirmations, not *time*.

Quote

> The probability of success [of a double-spend attempt] depends on the number of blocks [by which the honest network has an advantage], and not on the time constant T_0.
>
> — *Meni Rosenfeld; Analysis of hashrate-based double-spending (Meni Rosenfeld; 2012)*

In a traditional blockchain (e.g., Bitcoin, Ethereum Classic) confirmations occur, on average, at a predictable rate (that of the target block production frequency). Thus, for any *particular* traditional blockchain, a convenient time-based *rule of thumb* can be devised, e.g., a Bitcoin transaction is safe to accept after 1 hour. However, this approximation only works because blocks (and thus confirmations) are only produced locally (to that blockchain) and at a probabilistic (roughly constant) rate. Put another way, the frequency of confirmations is identical to the frequency of blocks, B_f Hz. Since $O(B_f) = O(1)$, the time-complexity of confirmation in these networks is also $O(1)$.

When using PoR, though, the assumptions behind that *rule of thumb* do not hold — while blocks on a single chain may be produced at a constant rate, that chain also gains a security benefit from other chains. For the case of a 2-chain simplex (where those chains have the same block production frequency), the rate of confirmations will be twice the rate of block production. This is easily generalized: for an N_1-simplex with simplex-chains that share some block frequency B_f, the rate of confirmation will be $\mathbb{C}' = N_1 \cdot B_f$ Hz. Thus, the rate of confirmations has complexity $O(\mathbb{C}') = O(N_1 \cdot B_f) = O(N_1) = O(c)$.

Let *confirmation time* be the duration breakpoint beyond which enough confirmations have occurred to consider a transaction *safe*. This is equivalent to the *rule of thumb* mentioned earlier. For a traditional blockchain, as mentioned, this is the product of some constant and the expected duration between blocks: ${B_f}^{-1}$. For a simplex, though, the expected *duration* is ${\mathbb{C}'}^{-1} \propto \frac{1}{N_1 \cdot B_f}$. Thus, as the simplex grows — as N_1 *increases* — the entire network's rate of confirmations also increases, and thus *confirmation time* approaches 0[64].

A 200-simplex with $B_f = 1/15$ has a confirmation rate of $\mathbb{C}' = 40/3 \approx 13.3$ Hz. An 800-simplex with $B_f = 1/60$ has the same confirmation rate. A 1400-simplex (the most optimized maximal simplex given *Amaroo's* initial configuration) with $B_f = 1/15$ has $\mathbb{C}' \approx 93$ Hz — $\sim$46.5$\times$ faster than EOS/Solana, $\sim$1116$\times$ faster than Eth2, $\sim$1400$\times$ faster than Eth1, and $\sim$55,800$\times$ faster than Bitcoin.

Note that PoR incents miners to publish blocks as soon as possible so that those blocks begin gaining reflections. If a miner does not publish a block immediately, then the reflections in that block become out-of-date very quickly as there are new, additional headers to reflect arriving constantly. Additionally, any competing block (published immediately by an honest miner) will begin acquiring reflections earlier, and contains more valuable reflected headers (incenting other miners to subsequently reflect it). So the published block has two distinct advantages over the withheld block. This mitigates the selfish mining[65] attack.

4.8 DoS and DAGs

Up to this point, simplex-chains have been treated like traditional blockchains, where each block has only one parent. Since the vast majority of a simplex-chain's security is provided by other simplex-chains (and only a small proportion comes from that chain's foundational consensus method), are attacks like an empty-block Denial of Service[66] (DoS) possible? If a simplex-chain were to use PoW, then it might be (relatively) trivial for an attacker to perform such an attack. This is because — in traditional blockchains — controlling more than 50% of the blocks produced provides *exclusive* control over *which candidate child blocks win* (i.e., are accepted into the canonical chain).[67] Is there a way that we can mitigate this risk? If blocks were permitted *more* than a single parent, can this *exclusivity* be denied?

4.8.1 Block-DAG Lineage

The idea that blocks in a chain can have multiple parents — i.e., the chain forms a Directed Acyclic Graph (DAG) that is not also a tree — dates back to (at least) late 2013 with the publication of GHOST[68] by Sompolinsky and Zohar. However, GHOST disallows multiple *canonical* parents, and a chain using GHOST defines its *canonical history* — the *main chain* — via the chain formed exclusively from the first parent of each block. A block's other, non-canonical parents are linked to *only* for the purpose of contributing to the total chain-weight.

In the two years after GHOST was published, a number of DAG-based blockchain designs were developed that facilitated merging histories from multiple parent blocks.

In mid-2014 I created a prototype DAG-based blockchain called Quanta[69] with a novel method of linearization that converged to a complete and stable ordering of blocks. This method was

[64]To say that confirmation time approaches 0 only tells the latter half of the process by which a transaction becomes confirmed. The first half of that process is *getting an initial confirmation*, which is effectively a small, but constant, overhead.
[65]See Majority is not Enough: Bitcoin Mining is Vulnerable by Ittay Eyal and Emin Gün Sirer.
[66]For an example of this attack, see Luke Jr's attack on Coiledcoin.
[67]Exclusive control of this nature also allows for protocol changes via soft-forks. Additionally, because these soft-forks can be undetectable (when done well) and don't necessarily affect the income of a miner substantially, bribery (of miners) is theoretically inexpensive. WRT UT, this problem is resolved in Section 4.8.3.
[68]Secure High-Rate Transaction Processing in Bitcoin (Sompolinsky, Zohar; 2013)
[69]Quanta source code, Quanta BitcoinTalk thread.

independently rediscovered[70] in mid-2015 by Lewenberg, Sompolinsky, and Zohar[71] — who also further developed and analyzed the method, which they named *the inclusive protocol*. Additionally, in 2016, Paul Firth further developed Quanta in his *Trustless Eventual Total Order* draft.[72]

In late-2015 several new alternative methods were also published, however, these are not generalizations of Nakamoto consensus. Namely: Lerner's DagCoin[73], and IOTA's Tangle[74]. Since then, multiple other models have been proposed, and some built.

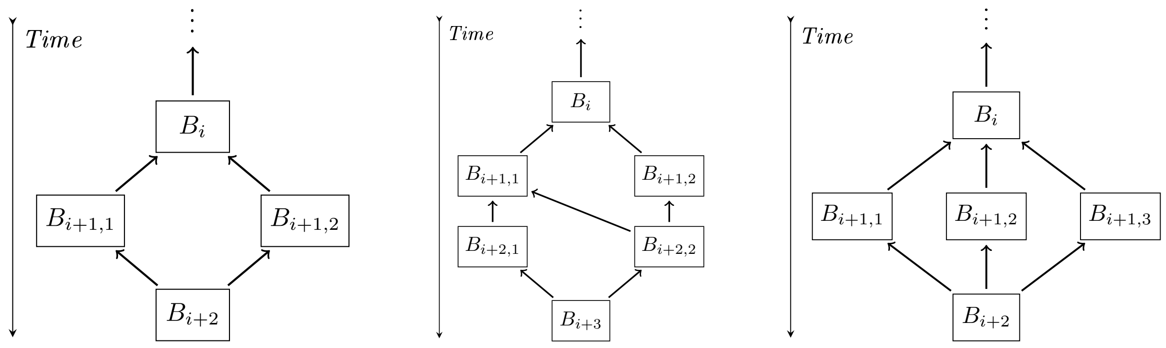
For the purposes of this paper, we are concerned with Quanta / the Inclusive Protocol (which is detailed in Inclusive Block Chain Protocols (Lewenberg, Sompolinsky, Zohar; 2015)).

4.8.2 Basic Structure

There are decisive advantages to using DAGs (instead of trees) as the fundamental structure of a chain. Namely, multiple histories (both compatible and incompatible) can be merged into a single, consistent history — a feature which eliminates stale blocks and thwarts attacks like an empty-block DoS. UT's simplex-chains must be block-DAGs to remain functional and avoid such DoS attacks.

Often, the motivation for using a block-DAG — instead of a block-tree — is to increase the block frequency. Since block-DAGs can reference multiple parent blocks, the stale-rate can theoretically approach (or reach) 0. In UT, increasing the block frequency eventually becomes counter-productive, though, since UT is sensitive to the size and number of headers that are produced (see Section 5.9). In UT, the purpose of using block-DAGs is to thwart certain attacks, not to increase the block frequency. The intention is for UT simplex-chains to use fairly typical block frequencies (e.g., 1 block per 15 s) — possibly decreasing those frequencies over time to increase capacity. Slower block frequencies also decrease the incidence of multiple parents — this matters because each parent typically increases the header size by 32 bytes.

Some basic block-dag segments are shown in Figure 22.



(a) A simple 2-parent example of a block-dag segment.

(b) A slightly more complex example of a block-dag segment.

(c) A 3-parent example of a block-dag segment.

Figure 22: Some examples of simple block-dag segments.

The essence of DAG-based consensus (at least the kind we're concerned with) is to *prioritize execution* of blocks and transactions based on the *security contribution* (i.e., weight) of each parent block (and that parent's ancestry). Based on a *most recent common primary*[75] *ancestor*, we can decide which parent's *history* has priority execution. Prioritized blocks are positioned *earlier* in the final ordering.

[70] As far as I can tell, the linearization methods produce identical results.
[71] Inclusive Block Chain Protocols (Lewenberg, Sompolinsky, Zohar; 2015)
[72] T.E.T.O Draft (Paul Firth; 2016)
[73] DagCoin Draft (Demian Lerner; 2015)
[74] IOTA BitcoinTalk Thread (Come-From-Beyond; 2015)
[75] In DAG (or DAG-like) chains, primary ancestors form the *main chain*, a.k.a. the *pivot chain*. Provided that parent blocks *are prioritized by cumulative work*, the chain of *primary ancestors* between a given block and the genesis block must be the *heaviest path* between the two.

When a block has more than one parent, the prioritized parent is the *primary* one. The chain of primary parents forms the *main chain*.

Term

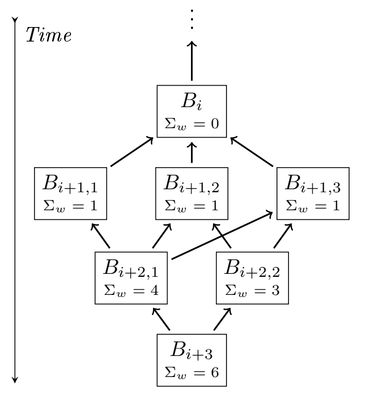
Main Chain: In a block-DAG, the *main chain* is the continuous chain of primary parents from the best block to the genesis block. Blocks that are part of the main chain are *on-main*, and blocks that are not are *off-main*.

Let's limit block-DAG blocks to two parents. If the best block has two parents, then each parent will have a *subgraph of blocks* between itself and the *most recent common primary ancestor* of the two parents. The subgraph which takes priority is that of the *prioritized parent's ancestry*. If that subgraph is a chain, then the ordering and execution of blocks is trivial. If it is not, then there must be another subgraph within that subgraph, and this algorithm is applied recursively to the prioritized parent. After the prioritized subgraph is processed, the remaining blocks (those that are only ancestors of the secondary parent block, if it exists) are ordered and applied — invoking recursion where necessary. Finally, the best block is applied. In this way, all blocks are executed after their ancestors, and there is a clear and total ordering that trivially converges.

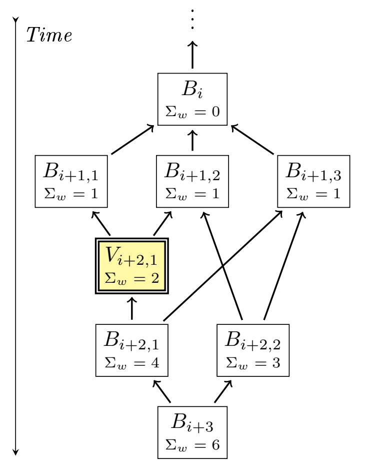
There is a simple generalization of the above to allow more than two parent blocks, too. That is: replace *all* but the last (worst) parent with a *virtual parent block* that links back to all remaining *actual* parent blocks (but contributes zero block-weight itself). Replacing the best parents (rather than the worst parents) with a virtual block means that the fork rule works when selecting the virtual block over the least-prioritized parent. This can be repeated to allow for arbitrarily many parents.

Figure 23 is an example of this algorithm for a moderately complex chain-segment ($B_i \cdots B_{i+3}$ which is 7 blocks total), and each step is enumerated and explained.

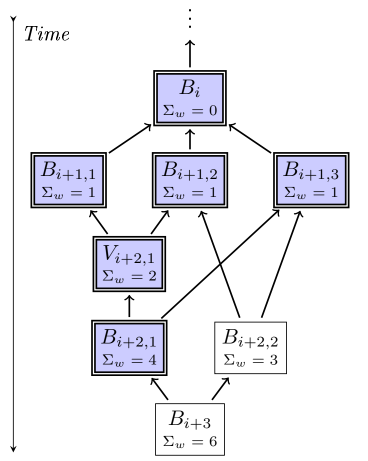
We should expect conflicting transactions (which might otherwise be attempted doublespends) to arise during this process. Ancestors of one parent may not be ancestors of another parent. The exact protocol for handling conflicts is up to the implementation, but there is no reason that secondary parents should be treated as invalid by default. We will discuss merging histories in Section 4.8.5.


Time
B_i $\Sigma_w = 0$
$B_{i+1,1}$ $\Sigma_w = 1$
$B_{i+1,2}$ $\Sigma_w = 1$
$B_{i+1,3}$ $\Sigma_w = 1$
$B_{i+2,1}$ $\Sigma_w = 4$
$B_{i+2,2}$ $\Sigma_w = 3$
B_{i+3} $\Sigma_w = 6$

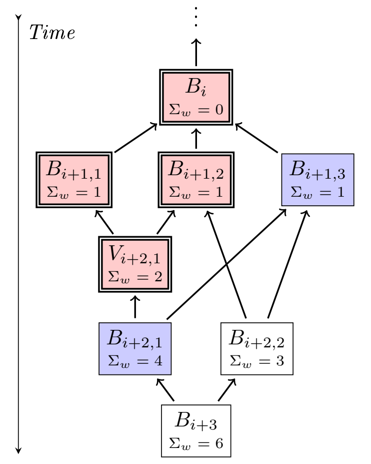
(a) How should we order this block-DAG? Note: children should *always* be after their parents, and *prioritization* means *earlier execution.*


Time
B_i $\Sigma_w = 0$
$B_{i+1,1}$ $\Sigma_w = 1$
$B_{i+1,2}$ $\Sigma_w = 1$
$B_{i+1,3}$ $\Sigma_w = 1$
$V_{i+2,1}$ $\Sigma_w = 2$
$B_{i+2,1}$ $\Sigma_w = 4$
$B_{i+2,2}$ $\Sigma_w = 3$
B_{i+3} $\Sigma_w = 6$

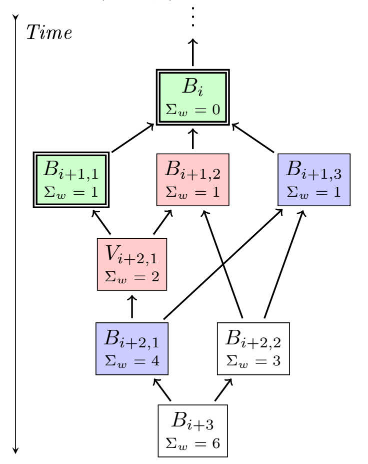
(b) The first thing we should do is create any virtual nodes that are required ($V_{i+2,1}$).


Time
B_i $\Sigma_w = 0$
$B_{i+1,1}$ $\Sigma_w = 1$
$B_{i+1,2}$ $\Sigma_w = 1$
$B_{i+1,3}$ $\Sigma_w = 1$
$V_{i+2,1}$ $\Sigma_w = 2$
$B_{i+2,1}$ $\Sigma_w = 4$
$B_{i+2,2}$ $\Sigma_w = 3$
B_{i+3} $\Sigma_w = 6$

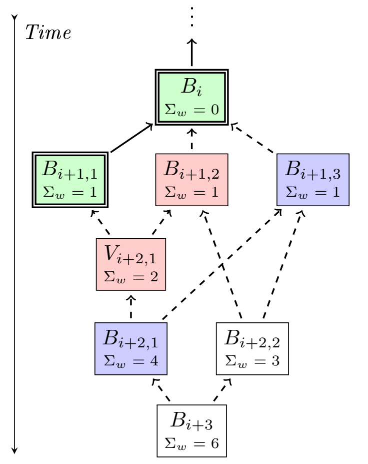
(c) Since there are two parents, we look at the *prioritized subgraph* (i.e., most worked).


Time
B_i $\Sigma_w = 0$
$B_{i+1,1}$ $\Sigma_w = 1$
$B_{i+1,2}$ $\Sigma_w = 1$
$B_{i+1,3}$ $\Sigma_w = 1$
$V_{i+2,1}$ $\Sigma_w = 2$
$B_{i+2,1}$ $\Sigma_w = 4$
$B_{i+2,2}$ $\Sigma_w = 3$
B_{i+3} $\Sigma_w = 6$

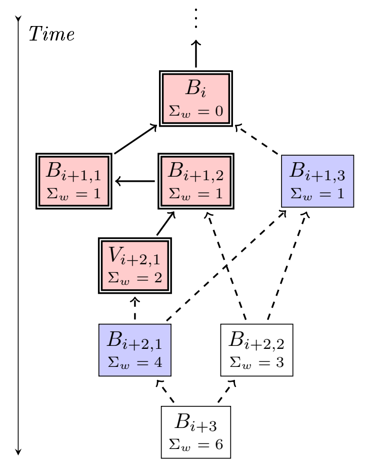
(d) Again, there are two parents, so we look at the *next* prioritized subgraph.


Time
B_i $\Sigma_w = 0$
$B_{i+1,1}$ $\Sigma_w = 1$
$B_{i+1,2}$ $\Sigma_w = 1$
$B_{i+1,3}$ $\Sigma_w = 1$
$V_{i+2,1}$ $\Sigma_w = 2$
$B_{i+2,1}$ $\Sigma_w = 4$
$B_{i+2,2}$ $\Sigma_w = 3$
B_{i+3} $\Sigma_w = 6$

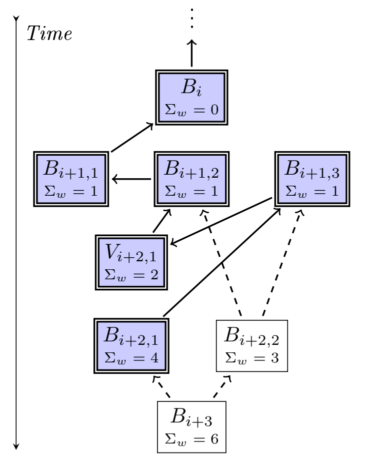
(e) We've found a chain. These blocks have the highest priority, so are executed first.


Time
B_i $\Sigma_w = 0$
$B_{i+1,1}$ $\Sigma_w = 1$
$B_{i+1,2}$ $\Sigma_w = 1$
$B_{i+1,3}$ $\Sigma_w = 1$
$V_{i+2,1}$ $\Sigma_w = 2$
$B_{i+2,1}$ $\Sigma_w = 4$
$B_{i+2,2}$ $\Sigma_w = 3$
B_{i+3} $\Sigma_w = 6$

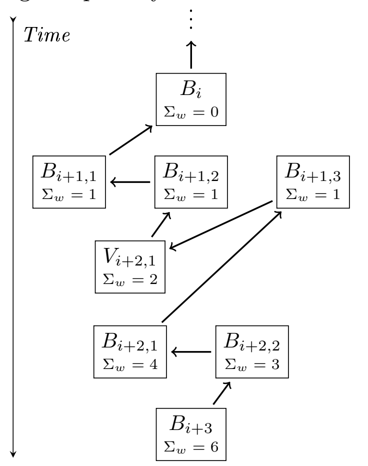
(f) We're now *ordering* the blocks — the solid arrows represent the final ordering. In this step we order the highest priority blocks.


Time
B_i $\Sigma_w = 0$
$B_{i+1,1}$ $\Sigma_w = 1$
$B_{i+1,2}$ $\Sigma_w = 1$
$B_{i+1,3}$ $\Sigma_w = 1$
$V_{i+2,1}$ $\Sigma_w = 2$
$B_{i+2,1}$ $\Sigma_w = 4$
$B_{i+2,2}$ $\Sigma_w = 3$
B_{i+3} $\Sigma_w = 6$

(g) Now that the highest priority blocks are ordered, we can order the *previous* subgraph.


Time
B_i $\Sigma_w = 0$
$B_{i+1,1}$ $\Sigma_w = 1$
$B_{i+1,2}$ $\Sigma_w = 1$
$B_{i+1,3}$ $\Sigma_w = 1$
$V_{i+2,1}$ $\Sigma_w = 2$
$B_{i+2,1}$ $\Sigma_w = 4$
$B_{i+2,2}$ $\Sigma_w = 3$
B_{i+3} $\Sigma_w = 6$

(h) Once more, we order the next-in-line subgraph.


Time
B_i $\Sigma_w = 0$
$B_{i+1,1}$ $\Sigma_w = 1$
$B_{i+1,2}$ $\Sigma_w = 1$
$B_{i+1,3}$ $\Sigma_w = 1$
$V_{i+2,1}$ $\Sigma_w = 2$
$B_{i+2,1}$ $\Sigma_w = 4$
$B_{i+2,2}$ $\Sigma_w = 3$
B_{i+3} $\Sigma_w = 6$

(i) And finally we order the last remaining blocks. (We could remove virtual blocks too).

Figure 23: Example: sorting a moderately complex block-DAG; note that the left parent is always the best parent, so will have priority. Each block is annotated with its *chain-weight* (Σ_w).

4.8.3 Block-DAGs Prevent DoS Attacks

Consider the situation where an attacker is attempting to deny service via the production of empty blocks, and that the attacker can create blocks faster than the honest network. Such a situation is illustrated in Figure 24. Since the goal of the attack is to prevent transactions from occurring, the attacker must[76] produce empty[77] blocks. Furthermore, if the attacker links back to any honest blocks then the honest blocks' histories will be merged with the canonical history; thus the attack would fail. The attacker's only available strategy is to produce a single chain of empty blocks.

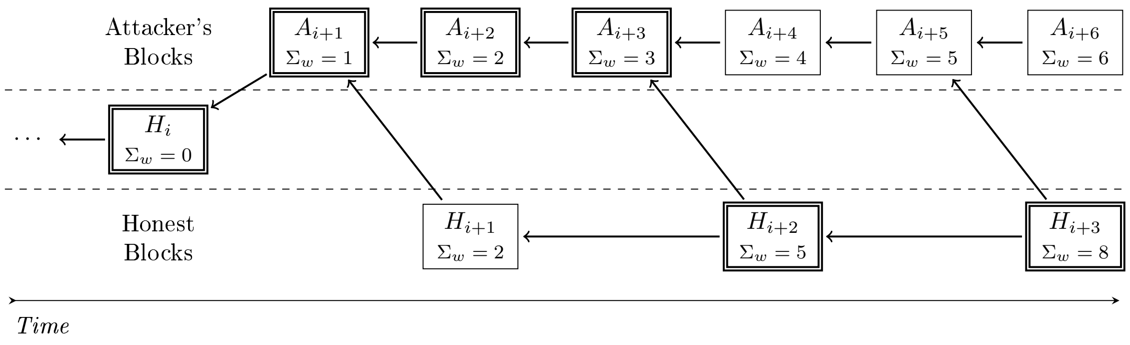


Figure 24: An attempted empty-block DoS attack on a block-DAG. The attacker's blocks, A_{i+x}, are mined in public and contain no transactions. Each block is annotated with its *chain-weight* (Σ_w); a double outline indicates that a block is part of the main chain. Even though the attacker produces 2× as many blocks as the honest network, the attack inevitably fails after a short while. Note: H_i is defined to have $\Sigma_w = 0$ for illustrative convenience.

The challenge of such a DoS attack is to prevent honest miners from extending the attacker's chain-segment. For traditional (non-DAG) chains — where each non-genesis block has exactly one parent — this is accomplished as soon as the attacker is able to reliably produce a heavier chain-segment than the honest network for a given period.[78]

However, if blocks are allowed to have *more* than one parent then there *is no point* where an attacker can *maintain* a DoS attack indefinitely. Instead, they can only *delay the execution* of some transactions for a short period of time. Particularly, if an attacker can produce $A_{\text{blocks}} > 1$ for every 1 block produced by the honest network, then the attack can delay honest transactions for up to approximately $({A_{\text{blocks}}}^2 - 1) \cdot {B_f}^{-1}$ seconds, where B_f is the frequency of block production (in Hz). After this (approximate) point, the weight of the honest chain-segment, which includes the attacker's chain-segment, is always greatest.

If an attacker performs a *repeating cycle* of these attacks, then it may be able to decrease the effective capacity of the chain by a factor of $\sim {A_{\text{blocks}}}^2$. The opportunity cost of this attack, for the attacker, is at least as much as the lost transaction fees.

We'll discuss DoS attacks again, shortly, in Section 4.8.7.

[76]The attacker could also fill the blocks with spam transactions. That's more work for the attacker, but also more work for the honest network (calculating and storing that state, maybe indefinitely). It's preferable that the attacker has minimal transactions in their blocks. It's tempting to think of ideas like: *since the attacker's blocks are empty, we can let honest nodes make larger blocks via some kind of weighted average block size calculation plus some flexibility in the size of blocks produced.* The problem with this is that it incents the attacker to fill their blocks with spam transactions, which is counterproductive.

[77]Note that the attacker should still be reflecting other simplex-chains so as to maximize the total weight of the attacker's chain-segment. Given this, the attacker's blocks will contain reflected simplex-chain headers but no transactions.

[78]For a traditional blockchain, this would also imply a 51% attack were possible, so this isn't a situation that those chains have to deal with. In a simplex, however, a miner can have > 50% of a chain's local hash-rate, so we need to handle this case.

4.8.4 A Criticism of GHOST

GHOST allows for blocks to link to a single canonical parent and multiple *uncle* blocks. In the full GHOST algorithm, uncle blocks contribute *weight* to the canonical chain-segment, but do not contribute *transactions.* Thus, uncles have *no ability* to substantially contribute to the canonical chain's *state.*

Consider an empty-block DoS against a chain using GHOST. If an attacker were to perform an empty-block DoS, the attacker could link back to honest miners' blocks as uncles, but never parents. Given this, there is no easy way for the honest miners to end or mitigate the DoS. The attacker can include honest miners' chain-work in a purely *beneficial* way — there is a symmetry, thus honest miners (and the network) are at the mercy of the attacker.

Why does this symmetry exist? Because the *cumulative weight* of each block (including uncles) is *divorced* from *the set of transactions* that is contributed by that block. However, with a full block-DAG, when an attacker links to uncles in this way, *they must allow for the execution of all non-conflicting transactions* (i.e., those which would not cause a doublespend to occur). Thus, GHOST *does not mitigate* empty-block DoS attacks; *only* a full block-DAG can do that.

4.8.5 Merging Histories

If a block B_{i+2} has two valid parents, how can we resolve this into one consistent history?

Consider the following subgraph, where squiggly arrows indicate secondary parents: Figure 25.

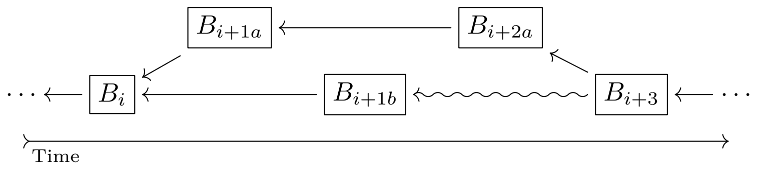


Figure 25: A simple block subgraph. Standard arrows indicate primary parents, and squiggly arrows indicate secondary parents.

By the ordering algorithm we covered above, we know that the blocks will executed in this order: B_i, B_{i+1a}, B_{i+2a}, B_{i+1b}, B_{i+3}.

Typically, a blockchain's state transition function can (at least partially) be verified via the pattern in Algorithm 7. We first load the parent's state, apply each transaction in sequence, and then calculate the new state root. If this matches the state root recorded in the block header, then the state transition is valid (all else being equal). This works fine for some blocks that have only one parent (e.g., B_{i+1a}, B_{i+2a}).

Algorithm 7 A basic state transition verification function

```
procedure VerifyStateTransition(parentStateRoot, TXs, stateRoot)
    midState ← LoadStateFromRoot(parentStateRoot)
    for tx in TXs do
        midState ← ApplyTx(midState, tx)
    end for
    return stateRoot == GetRoot(midState)
end procedure
```

We *can* use this algorithm *from B_{i+1b}'s perspective* to ensure B_{i+1b} is internally consistent; however, we *cannot* use it with the above subgraph.

Why? Because B_{i+1b}'s parent is B_i, but the block immediately executed prior to B_{i+1b} was B_{i+2a}. Thus, the algorithm doesn't work for B_{i+1b}: we need to use a different prior state, and therefore the *resulting* state root *will not match* the state root in B_{i+1b}'s header.

Moreover, the algorithm doesn't exactly work for B_{i+3}, either. In B_{i+3}'s case, its primary parent is B_{i+2a}, but it's state root should be the result of applying B_{i+1b}'s transactions to B_{i+2a}'s output state, and then applying B_{i+3}'s transactions to that mid-state.

It is clear that the algorithm breaks for B_{i+3} because it has more than one parent.

The reason that it breaks for B_{i+1b} is more subtle: B_{i+1b} is not part of the *main chain.*

Observe that the pattern works for on-main blocks with a single parent, but not for off-main blocks, nor for on-main blocks with multiple parents.

The solution is to combine new off-main ancestors (B_{i+1b}) with the relevant on-main descendant (B_{i+3}) and process them as a single operation. In principle, we're collapsing the full block-DAG down to just the main chain, so that traditional blockchains' state-transition logic works for block-DAGs, too.

This has two important and related implications. First: transactions in off-main blocks may not be valid at execution time, because the previous state root changes. Second: only the state roots of on-main blocks are accurate and reliable. We should be careful to verify proofs against *only* on-main blocks — methods like SPV *are not safe* against off-main blocks.

We can now describe the execution in terms of the main chain by grouping novel off-main ancestors with the next main chain block:

$$\begin{tikzcd}
\cdots \arrow[r] & \boxed{B_i} \arrow[r] & \boxed{B_{i+1a}} \arrow[r] & \boxed{B_{i+2a}} \arrow[r] & \boxed{B_{i+1b}} \arrow[r] & \boxed{B_{i+3}} \arrow[r] & \cdots
\end{tikzcd}$$

Execution Order

The dotted boundary around ($B_{i+1b} \to B_{i+3}$) indicates that the blocks are processed as a group. The dashed boundary around B_{i+1b} indicates that it is, in a sense, *virtual* within the state-transition function. B_{i+1b} is more like a part of B_{i+3} than a block in its own right.

See Figure 26 for an addition example.

If we want to support stateless full nodes (and fraud proofs), this grouping strategy adds a constraint to the block structure. Particularly, the complete witness structure of B_{i+3} is not simply the witnesses for B_{i+3}'s transactions combined with the witnesses for B_{i+1b}'s. Instead, it is a new witness structure initialized from B_{i+2a}'s state, and must cover *both* blocks' transactions.

In regards to fraud proofs, the technique that we discussed in Section 4.5 requires additional data to be included in B_{i+3}. Recall that the transaction root should correspond to a merkle tree with leaves alternating between state roots and transactions, with each root being that of the state immediately prior to the transaction's execution. Thus, B_{i+3} must provide intermediate state roots that uniquely correspond to each transaction in each novel off-main ancestor. Furthermore, the transaction root should commit to *all* of these {mid-state root, transaction} pairs, not only those of transactions that were included directly in B_{i+3}.

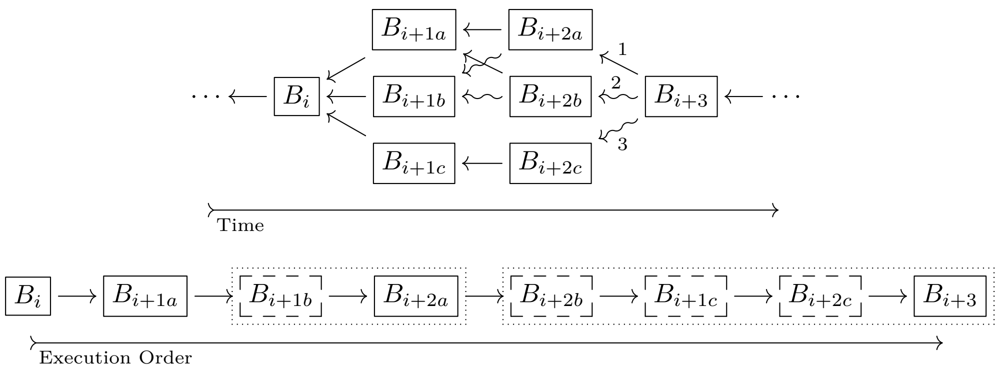


Figure 26: A complex block subgraph and corresponding execution order. In the subgraph, standard arrows indicate primary parents, and squiggly arrows indicate secondary parents. In the execution order, dotted boxes indicate groups of blocks that are executed together — when an on-main block has multiple parents, the novel off-chain ancestors (indicated by a dashed outline) are executed as part of the on-main block's execution.

4.8.6 Adapted NIPoPoWs for Block-DAGs: NIPoPPoWs

Warning

Reminder: Section 4.8.6 and sections on NIPoPoWs are speculative.

Recall that in Section 4.6.5.2 we noted that block-DAGs are currently incompatible with NIPoPoWs. This is because the traversal method of NIPoPoWs does not know anything about whether a block is on-main or not. Additionally, the fork rule does not (and should not) prioritize blocks based on μ values. So we should expect that some off-main blocks have a higher μ value than their on-main sibling and/or their first on-main descendant. Therefore, currently, a valid NIPoPoW can lead to an off-main block — i.e., a block that is not useful for SPV.

We do not have a problem traversing via off-main blocks, though. They share the same cryptographic properties (with regards to interlinks) as any other block.

So, our major goal is to find a way to always arrive at on-main blocks. A valid block-DAG NIPoPoW should never terminate at an off-main block.

One problem is that, when a block is created, the network does not know if it will be on-main or not. Future blocks determine whether it is on-main or not.

Furthermore, we cannot be certain that all future blocks will agree that any specific block is on-main or not. We should expect that all far-future blocks will agree, but how long should we wait and is it reliable?

Could we simply modify the interlink structure to add a flag for off-main blocks?

No — this is not sufficient. There is only one way (in principle) to always arrive at an on-main block: via ever more recent on-main blocks. It is only because of an on-main block's *posterity* that it becomes (and remains) on-main. Simply flagging off-main blocks does not work because, once we reach an off-main μ-superblock, we know nothing about which prior blocks are on-main or not. An off-main block is (on its own) never a reliable source of prior blocks' on-main status.

Here is how we can reliably arrive at on-main blocks: first, starting from all chain-tips, we find the most recent common on-main ancestor. If this is very close to the chain-tips, we can, instead, use an on-main ancestor a few confirmations back — so we can meet any particular security parameter we might choose. We can be confident that this ancestor is on-main, regardless of which chain-tip eventually ends up as the on-main block. We must then follow the chain of primary parents.

Therefore, the interlink structure *must* include enough information to traverse the μ-superDAGs via on-main ancestors, only. Just because we *can* traverse via on-main blocks only does not mean that we will, though. In fact, we will traverse via two paths: one normally, and the other via on-main blocks only.

To support this, we'll modify the interlink structure such that each link has an optional second component: the most recent on-main ancestor at that μ level. Most of the time, we should expect blocks to be on-main, so there will only be one interlinked entry at that μ level. However, in the case that a μ-superblock is off-main, the interlink entry is both that off-main μ-superblock, and the most recent on-main μ-superblock (which is necessarily older than the off-main μ-superblock).

Thus, we effectively have two μ-superDAGs: one being the *best-blocks* μ-superDAG, and the other being the *on-main only* μ-superchain. (Each block in a block-DAG has only one primary parent, so a chain-tip's primary ancestry really is a chain, not a DAG.)

We then construct the main suffix proof for two cases: the *best-blocks* suffix proof (regardless of on-main status), and the *on-main only* suffix proof. These two proofs will always be consistent, though the best-blocks proof may not include all μ-superblocks that are included in the on-main proof. To account for this, we can require that, for each μ level, the best-blocks μ-superDAG suffix subgraph is a superset of the corresponding on-main μ-superchain suffix. (That is: we make sure the necessary on-main blocks are included in the proof if they wouldn't be otherwise.) Both together comprise the suffix proof.

Regarding infix proofs, we can use the existing[79] `followDown` algorithm, is used to navigate between superblocks and regular blocks, with only one modification: an additional parameter determines whether `followDown` should traverse the *best-blocks* path to the block, or the *on-main only* path. To generate proofs, we run `followDown` over each path. We similarly modify the `verify-infx` (*sic*) and `ancestors` functions used to verify infix proofs.

It is worth noting that the on-main proofs alone are not sufficient for secure NIPoPoWs: in some cases the on-main only proofs will imply substantially less total chain-weight than the best-blocks proofs. For example: if only 50% of blocks were on-main, then the resources an attacker needs to generate a competing NIPoPoW (for the on-main proofs) is only 50% of what they would need compared to a best-blocks proof. Thus, we prioritize one NIPoPoW over another based on the *best-blocks* proofs (rather than the *on-main* proofs). We do, however, validate that the target of the NIPoPoW is on-main via the accompanying on-main proof. The on-main path is only inconsistent with the best-blocks path in the case that the target block is off-main.[80]

A major difference between the best-blocks and on-main proofs should not be an issue for UT, though, as UT uses modest block frequencies for each chain. We should expect blocks to have a single parent most of the time. (Using the logic from Section 4.6.1, we'd expect simultaneous blocks ~2% of the time — give or take.)

As a final note, what happens when there are multiple paths in a μ-superDAG? Note that this situation only occurs in the best-blocks proof. Since the best-blocks proof is not concerned with the posterity of blocks (their on-main status), there is no issue here. We can choose between the paths by whichever means we like — e.g., picking the block with the lowest PoW, or picking on-main blocks over off-main blocks when they are siblings in the same μ-superDAG.

We've just sketched the construction of a new type of NIPoPoW that can prove *both* the full chain-weight and the posterity of blocks. The best-blocks half of the proof is almost a regular NIPoPoW on its own, but the other half of the proof (traversing on-main blocks only) is not independently secure. However, *together* they are secure.

[79] `followDown`, `verify-infx` and `ancestors` are functions defined in the Non-Interactive Proofs of Proof-of-Work (Kiayias, Miller, and Zindros; 2018) paper.

[80] Note: when traversing the best-blocks μ-superDAG, we *do not care* about the status of on-main blocks. Thus, the on-main interlinks in off-main μ-superblocks are not important, and we should not count any differences there as inconsistencies with the on-main μ-superchain. Similar to NIPoPoWRs, the best-blocks proof only proves that a block *exists*, and no conflicts are possible via different paths.

We should distinguish between this new kind of NIPoPoW and the regular kind. The main difference is that what we have just sketched proves *work and posterity*, rather than just work. Thus, it is a *Non-Interactive Proof of Posteritorial*[81] *Proof of Work* (NIPoPPoW).

4.8.7 Block-DAG Interaction With The PoR Graph

Up to now we have mostly considered block-DAG's in isolation; outside UT's broader context. Particularly, we should ask: *do the Axioms of Availability and Maximal Reflection change the game?*

Recall that, when defining the Axiom of Maximal Reflection, we made sure to recursively include blocks' ancestors, not just reflections. This was for two reasons. First, L_i should not explicitly reflect its own ancestors — links to parents *are* reflections, but are the trivial case where the local and reflecting chains are the same, and the local and reflecting blocks are the same. Second, L_i should not skip accounting for R_k's implied reflection of its parent(s), so we treat parents like any other reflected block — we must be sure not to leave any gaps.

There is still one gap, though: L blocks that are not an ancestor of L_i.

It should be clear that parent links are qualitatively different from reflections: child blocks attest to the validity of their parent blocks, and to continuity between these two blocks. The implicit 'PoR' between these blocks is also free, unlike explicit PoRs between blocks of different chains.

Unlike traditional chains, UT blocks are not only allowed to have multiple parents, but *can always include a valid chain-tip as an ancestor.* Recall the Axiom of Maximal Reflection's principle: *A miner must reflect all possible blocks.* Considering the symmetry between parents and reflected blocks, we can draft a new network principle: *A miner should build on all possible chain-tips.*

In effect, we want to ensure that a dishonest miner cannot evade the contradiction in refusing to build on a valid chain-tip whilst acknowledging it as available. If it is unavailable then the miner should not reflect the blocks that imply it is available; and if it is available and valid then it should be a parent of their draft block.

So far, this principle is convergent with honest miners building on *valid* blocks — including every chain-tip as a parent is what we expect honest miners to do by default. However, what are honest miners to do if there is an invalid block? We need to consider this because a block with a valid PoRs sub-block does not necessarily have a valid transactions sub-block.

We have already discussed the solution in Section 4.5: link back to, or *flag*, that block as invalid *with an attached fraud proof*, and never use that block as a canonical ancestor. The transactions in that block aren't important — excepting the offending transaction, which is part of the fraud proof. And we can reconstruct the PoRs sub-block using $O(1)$ data thanks to Maximal Reflection. Thus, an invalid block is only as large as its header, list of reflected PoR graph-tips, and fraud proof. That's $O(\log c)$ in the worst case.

4.8.7.1 Axiom of Unified Ancestry

This is all that we need to plug the gap: ensure honest miners can always build on *any* local chain-tip, valid or not. If the chain-tip is valid, the honest miner uses it as a parent. If it is invalid, the honest miner flags it as invalid. It is still an ancestor cryptographically, but not with respect to its transactions and chain-state.

[81] A definition is required as this word is an almost-neologism:
Posteritorial *adjective*: having posterity; supported by its descendants, especially regarding well-known descendancy, epistemic antecedence, root-causal nature, etc; having a prototypical and defining quality that was widely inherited.
Aside: Perhaps curiously, a prior candidate modifier was the latin phrase *per posterās* — literally 'through what came after'. But also faux-translated as '[as] per post eras', which is roughly equivalent to the English phrase 'via posterity'. A downside of *per posterās* is that, phonetically, it's very similar to 'preposterous', and Non-Interactive Proofs of Preposterous Proof of Work is not an apt name.

Axiom

Axiom of Unified Ancestry

Principle: A miner must use all valid chain-tips as parents, and flag invalid ones.

Predicate: A block, L_i, is invalid unless, for all reflected blocks R_k:

i. R_k is valid under this axiom; and
ii. each L block reflected by R_k:
   a. is an ancestor of L_i; or
   b. is flagged as invalid by L_i; or
   c. was flagged as invalid by an ancestor of L_i.

This new axiom changes the game for block-DAGs.

4.8.7.2 DoS Attacks

Previously, an attacker could delay transactions for $\sim{A_{\text{blocks}}}^2$ many blocks because they could evade the contradiction in the PoR graph. Now, they cannot. Thus, empty block DoS attacks against simplex-chain block-DAGs are limited to $\sim A_{\text{blocks}}$ in length (on average).[82] In terms of DoS potential, this is *exactly* the same as if the simplex-chain were a traditional chain and the attacker always built on the best block, but only made empty blocks. Now that we've added the Axiom of Unified Ancestry, the alternative for the attacker is producing only invalid blocks.

Aside

Note that this axiom applies and is verified in the context of the PoR graph (not chain-state).

4.8.8 NIPoPPoWs + NIPoPoWRs

Warning

Reminder: Section 4.8.8 and sections on NIPoPoWs are speculative.

In Section 4.8.6 we devised NIPoPPoWs, modified NIPoPoWs that work with block-DAGs by accounting for posterity. That construction depended on a *best-blocks* proof in combination with an *on-main only* proof. In the case of simplex-chain block-DAGs, we do not want to use the same best-blocks proof since it only accounts for *local* chain-work and does not account for reflections.

Instead of a standard best-blocks NIPoPoWs, can we use NIPoPoWRs from Section 4.6.5.2? Functionally, both serve the same purpose: proving that a block *exists*.

One potential problem is that the NIPoPPoW best-blocks proof substantially overlapped with the on-main only proof — that isn't the case for NIPoPoWRs as *most* of the μ-superblocks will be from reflecting chains. Rather, we expect the best posteritorial μ-superchain to have a μ level approximately $\log_2(N_1)$ lower than that of the best PoR μ-superDAG (since there are $N_1\times$ as many blocks). In other words, only $1/N_1$ local blocks are at the same μ level as the highest μ-superDAG.

Let us take stock for a moment. Each axiom adds a constraint on what the simplex does consensus on, and each block's relationships to other blocks. What are those constraints on consensus created by our current axioms?

- Availability: the availability of blocks.
- Maximal Reflection: the order of blocks, and that there are no 'gaps' in the PoR graph.
- Unified Ancestry: the ancestry of blocks, and that there are no 'gaps' in each simplex-chain.

These are properties that are verifiable *simplex-wide* — as part of tracking the PoR graph, each node can reconstruct each simplex-chain's *local* interlink structure.[83] *Thus, the PoR graph knows the last on-main block for each simplex-chain.*

[82] We will reduce this further in Section 4.9.

[83] It might go without saying, but an improper interlink structure is a reason for a block to be invalid, so bad interlinks will be rejected by the PoR graph.

So, can we use the same technique as we used before — tracing two proofs? Well, we will need to modify the interlink structure once again, but it seems in-principle possible.

Compared to NIPoPPoWs, we have greatly restricted the possibilities regarding to on-main blocks: Unified Ancestry means that *both* on-main and off-main blocks contain reliable[84] information regarding posterity. If a block has only one parent, then that parent must be on-main; off-main blocks need to use all available chain-tips as parents. If child blocks have only one parent, then the block is on-main. If the block has multiple parents, then one must be on-main, and we can always find the most recent common primary ancestor of those parents to find an on-main block. If child blocks have multiple parents and share the same primary parent, then we can swap to the on-main sibling whenever needed. If we get stuck, we can repeat any of these processes multiple times to increase our knowledge of that chain-segment — this means we can always determine on-main status via the PoR μ-superDAG. To prove posterity, we just need to show a large enough segment of the block-DAG so that we can, *with certainty,* identify which blocks within it are on-main. From there, determining on-main status of ancestors is trivial.

So we no longer need two proofs, but we do need a bit more information than a stand-alone NIPoPoWR.

Since this method is almost the same as our previous NIPoPoWR method, we'll avoid creating a new acronym and just re-use the term 'NIPoPoWR' to refer to this version, too.

Aside

Note: there is plenty of excess capacity available to us if the above is insufficient or insecure. We could, for example, use a NIPoPPoW to prove some on-main block, then prove *each* high level μ-superblock in the NIPoPPoW via NIPoPoWRs.

There is one other fact that we need to prove: that no fraud proof was ever recorded for the target block.

4.8.8.1 Integrating Fraud Proofs

NIPoPoWs (in general) exist in a unique context: we expect that a lot of time has passed since the target infix block was mined.

If PoR sub-blocks keep track of a set of *all* fraudulent blocks (and this is enforced), then we can use a much later block (from any chain) to prove whether any fraud proof for any particular block was ever recorded. We don't care what the fraud proof actually was, since the PoR graph already came to consensus on the fact it exists. This can be done via a Merklix tree, or any other merklized structure that allows us to prove non-membership. Therefore, we can use one of the intermediate μ-superblocks to prove that no fraud proof was recorded for the target infix block. Since a long time has passed (at least some security parameter of our choosing), we can be confident that no fraud proof was possible, and thus that the block is valid.

We could also provide additional proofs (of non-membership) for e.g., all blocks used in the complete full NIPoPoWR, if we cared to. Blocks with invalid interlinks cannot be used as superblocks, though, so additional proofs should not be necessary.

This completes the sketch-construction of NIPoPoWRs.

4.9 Lowering Block Production Variance

Is it possible to *dramatically* lower the variance of block production in PoW blockchains without altering incentive structures, compromising security, or changing the probability of generating a valid block?

Yes. The method relies on the *structure* of the network, rather than the consensus protocol itself. Particularly, the network must be structured such that miners' choices result in decreased block production variance — an emergent phenomenon. It's important that it is emergent and not

[84]It's not 100% reliable, but it's good enough that we can always find an on-main block without much effort.

synthetic (e.g., by increasing the block reward with time-since-last-block) because we don't want people to game the system. It's better to have a simple system with emergent properties than a complex system with those properties "designed in".

Say you have a network with 10 chains: $C_0, C_1, C_2, ..., C_9$. If the networks are separate, then you have 10 groups of miners: $M_0, M_1, M_2, ..., M_9$. They have to choose one chain to mine on, so the distribution of miners is expected to be approximately the distribution of normalized block rewards plus tx fees. The proportions of block rewards between C_i & C_j don't really matter, we expect the mining groups M_i & M_j to just sort themselves out due to market forces. For simplicity, though, this example assumes that mining rewards and the distribution of miners is an even 10% across the board.

If the network has spare capacity (i.e., transactions are mostly cleared out with each block; the mempool for each chain is ~empty) then we have a situation like this:

Set $t = 0$ immediately after a block is published on a chain. Then, as t progresses, transactions with fees should build up in the mempool, so $\text{TxFees} \propto t$. The reward for mining a block is $r + \text{TxFees}$ for some block reward, r.

The potential reward-over-time for a miner (t vs $r + \text{TxFees}$) looks like a sawtooth function with a y-axis offset. It builds as more transactions pile up, and drops back to the baseline reward after a block.

If the miners $M_0, ..., M_9$ are capable of working on one of any $\{C_0, ..., C_9\}$ (and they have identical ROI profiles to the other miners), then they're incented to work on the chain with the most transactions in the mempool. That means: miners should, roughly, work the chain that has gone the longest without a block. What should we expect based on those incentives? Miners should work on each chain only in the final moments of the block production cycle. If block times were set to 60s, then they'd start mining at around the 54s mark because that's how they maximize their ROI.

Why wouldn't they just keep mining on the same chain? Because in the time that they focused on one chain, another one passed that >54s high-ROI threshold and thus has the best ROI potential per hash done.

We should thus expect that this configuration of chains actually *synchronizes* miners, resulting in block production that is somewhat regular and lower in variance.

Term

> **Miner Resonance**: The effect whereby block production *variance* is reduced when miners can (and do) collectively change which chain they are currently mining faster than blocks are produced for those chains, due to changes in network-wide incentivization.

The average hash-rate on each simplex-chain, as described above, is always the same regardless of which of the two miner strategies are used. However, the variance of block production on each of these chains won't be that of a chain with 60s block times, it'll be closer to that of a chain with 6s block times.

4.10 Simplex Security and the Confirmation Equivalence Conjecture

4.10.1 Simplex Security

Simplexes are secure if attacking a single simplex-chain is as difficult as attacking the whole simplex (the network).

Say the attacker controls q proportion of the network's work-generation capacity (e.g., hash-rate for PoW chains), and honest nodes control p proportion, such that $p + q = 1$. An attacker with $q < p$ can attempt doublespends, and should succeed some of the time (see Section 4.10.2), but never with certainty. The attacker should only be able to *control* the network's history if $q > p$.

Successfully attacking a single blockchain requires an attacker to publish an alternate chain-segment that the fork rule considers heavier than the corresponding public chain-segment. Those two segments will have a common ancestor, so the weight of the attacker's segment must be greater than the weight of the public chain-segment (both start at that common ancestor).

Simplex-chains evaluate block-weight as the sum of work done on that block, plus work done on reflecting blocks. So effecting a doublespend on one simplex-chain requires generating a chain-segment with more total block-weight (including reflections) than the public chain-segment.

It is not viable for the attacker to mine the attacking chain-segment in public (it is self-defeating; see Section 4.8), so they must mine it in private. Blocks mined in private will not gain reflections from honest miners on other simplex-chains, so any reflections contributing to the doublespend must be created by the attacker. The attacker's reflecting blocks (which reflect the attacking chain-segment) on other simplex-chains can be public, but the attacker's task is simpler and not detectable if the attacker mines reflecting blocks in private.[85] There is no viable way for the attacker to prevent honest miners from reflecting the honest chain-segment, including new blocks that extend it, nor to prevent the honest chain-segment from reflecting blocks from other simplex-chains (again, see Section 4.8).

The honest chain-segment, on the targeted simplex-chain, will not be reflected by the attacker (since that would be self-defeating). Similarly, the attacker's chain-segment will not be reflected by the honest network, since it's being mined in private.

Let r be the network-wide rate-of-work (hash-rate). Over some attack duration d, on *average* we expect the attacker's chain-segment to weigh qrd and the honest chain-segment to weigh prd. Thus, for the attacker's chain-segment to *reliably* win, it must be that $qrd > prd \implies q > p$. ■

4.10.2 Confirmation Equivalence Conjecture

Term

Confirmation Equivalence Conjecture (CEC): The conjecture that, when using PoR and appropriately converting work, confirmations of reflecting chains can be treated as *equivalent* to local confirmations of the same weight. See Equation 33.

The Confirmation Equivalence Conjecture is intimately connected to why PoR works. If it were false, then PoR could not work, and neither would simplexes. Additionally, this identifies the root of UT's $O(c)$ confirmation rate — the conjecture implies that confirmation rate is proportional to the number of mutually reflecting chains.

Can we test it? If so, how?

Consider a traditional PoW blockchain (e.g., Bitcoin, Eth1, etc). It's well known that the risk of a doublespend against a particular block is related to the number of *confirmations* that block has.[86] However, this is not a linear relationship; rather, it is similar to *exponential decay*. After the first few confirmations, each additional confirmation reduces the risk of a doublespend by approximately the same factor.[87] So *security takes time*, because confirmations take time.

Now, consider a simplex: the equivalence conjecture says that it doesn't matter whether confirmations come from the *local* chain, or if they come from a *reflecting* chain. So, a simplex-chain in a 2-chain simplex *should*, after C *local* confirmations, be as secure as a standalone blockchain with $2C$ confirmations. This is true for larger simplexes, too: *doubling* the size of a simplex means we can *halve* the number of *local* confirmations that we wait for — the chance of a doublespend succeeding should be approximately the same. More generally, a simplex-chain in an N-chain simplex will, after C *local* confirmations, be as secure as a standalone blockchain after CN confirmations.

This is something we can test! (See Section 4.10.4 for that.)

As blockchain architects, if the CEC is true, then UT allows us to maintain security by trading *waiting time* for *more simplex-chains!*

[85] Additionally, if reflecting chains maintain projections as headers-only chains (i.e., construct a main chain via SPV rules), then it should be much harder for an attacker to succeed if reflections are mined in public.
[86] Analysis of hashrate-based double-spending (Meni Rosenfeld; 2012)
[87] The exact factor depends on the attacker's hash-rate as a proportion of the network's (i.e., q).

Aside If you were looking for an essential aspect of how UT breaks the core conflict of the Trilemma…

4.10.3 Generalizing Doublespends

With a traditional blockchain, the network has no "memory" of work (mining) that has been done which did *not* result in a valid block (i.e., the main chain does not record or account for that work). Specifically, the network gains no knowledge of how much work has been done *between* blocks. This is almost self-evident: each block is the *sole* way that work can be *added* to the chain. Naturally, there is no smaller unit of contributed work than an individual block — so there's no "memory".[88] When an attacker publishes a heavier chain-segment which causes a reorganization, it is *immediately* in the interests of miners to begin building on the attacker's best block. A heavier chain-segment is *decisively better* in all cases. That may seem so obvious that it isn't worth mentioning, but it's actually a *specific case* that only works for traditional blockchains. If there were some "memory", it might be worth miners *continuing* to mine their *existing* block, *instead* of reorganizing and building on the attacker's best block.

What kind of "memory" could there be? Are *Proofs of Reflection* a "memory"?

As it turns out: yes.

Let the total chain-weight of the attacker's best block be called W_A, and the total chain-weight of the honest network's best block be called W_H.

Consider a miner of a non-DAG simplex-chain (chain L) who is working on a *draft block* during an active attack (though, of course, the miner does not know about the attack). Specifically, let's consider *just before* the attacker publishes their chain-segment. In the memory pool[89] of that miner, there will be PoRs, and those PoRs are *only* valid for the *honest* chain-segment (most likely they are for the best known honest block, but don't need to be). Each L block, in and of itself, only adds $\sim w$ weight to the L chain. However, if that *draft block* contains $\sim n$ pending PoRs, and each PoR provides $\sim w$ weight on average, then a valid block (created from that draft) will contribute, via PoRs, $\sim nw$ weight to the *honest chain-segment.* What happens when the attacker's chain-segment is published (along with their reflecting chain-segments for other simplex-chains)? The miner has a choice: reorganize to the attacker's chain-segment and mine on top of a chain with weight W_A, *or* continue mining on the honest chain-segment that weighs $W_H + nw$.

That extra term, nw, is the "memory" we're looking for, and it makes all the difference. It is only safe for the attacker to publish their chain-segment *after* $W_A > W_H + nw$ — only then is it *decisively better* than the honest chain-segment. If, however, the attacker publishes their chain-segment when $W_H < W_A < W_H + nw$: the best chain-segment for honest miners to mine is *still* the honest chain-segment, not the attacker's!

For non-DAG chains, miners don't actually have to change their behavior at all: their choice is the same, whether they are mining a simplex-chain *or* a traditional chain. That choice is: *of all possible blocks to mine, which results in the greatest reward?* If they build on the honest chain-segment, does their resulting block have more total chain-weight than if they were to build on the attacker's chain-segment?

There are some subtleties to consider if chains are DAGs or use GHOST: both chain-segments can be included as ancestors, but which should take priority? If the weight of PoRs is not taken into account (or applied after evaluating the order of parents), then the attacker's chain-segment takes priority when $W_A > W_H$. This rule conflicts with what we discussed above, so it must be that parents are ordered based on their chain-weights *including* any PoRs in that block — the honest

[88] We're not concerned with things like *weak blocks* or super-block hybrid designs; though it's possible that doublespend methodology for those designs has some overlap with this section.

[89] The *memory pool* is where unconfirmed transactions (and other things) are stored before they are included in a block. Each node has their own memory pool. A miner typically decides a block's contents by choosing the combination of transactions from their memory pool that results in the largest reward (i.e., transactions with the largest fees).

chain-segment should take priority when $W_A < W_H + nw$. Also, note that this is why the weight of PoRs is attributed to the *reflected* block (usually a parent) rather than the block that contains the PoRs. In essence, the inclusion of this "memory" requires that the fork rule take *context* into account — particularly the context of the descendant block. This is not a paradox because the fork rule is *only* used to choose the priority chain in the context of some descendant block (even if that block is an invalid draft).

Aside In the context of the longest PoR chain, this property is made somewhat obvious.

4.10.4 Testing the Confirmation Equivalence Conjecture

Let's test the Confirmation Equivalence Conjecture.

4.10.4.1 Hypothesis

The hypothesis is that: doublespends against an *individual simplex-chain* in an N-chain simplex after $\mathcal{C}$ *local* confirmations are *as hard* as doublespends against a traditional blockchain after $\mathcal{C} \cdot N$ confirmations.

First, define $P(q;c)$, where c is the number of confirmations and q is the proportion of global hash-rate controlled by the attacker, to be the function that returns the probability of an attempted doublespend succeeding on a traditional blockchain according to Rosenfeld's analytical solution[90]:

$$\forall c \in \{1,2,\dots\} : P(q;c) = \begin{cases} 1 - \sum_{m=0}^{c} \binom{m+c-1}{m} (p^c q^m - p^m q^c), & \text{if } q < p \\ 1, & \text{otherwise} \end{cases} \tag{30}$$

Next, we will declare a new function, P', and assume that *it exists and is measurable*. Particularly: $P'(q; c = \mathcal{C}; N_1 = N)$ is the probability of a doublespend attack succeeding against a simplex with N simplex-chains after $\mathcal{C}$ confirmations, given an attacker with q proportion of the network hash-rate. We do not know the definition of this function, but we can measure its output.

As a base case, it is taken that:

$$P(q;c) \equiv P'(q;c;N_1 = 1) \tag{31}$$

Note that a simplex of 1 chain (the 0-simplex) *is* a traditional blockchain. So we can take Bitcoin for example: the probability of an attack succeeding after 6 confirmations is given by $P(q;c=6) \approx P'(q;c=6;N_1=1)$.

One prediction of the hypothesis is that the probability of a successful attack against a simplex with N simplex-chains after $\mathcal{C}$ confirmations is equal to that of a successful attack on a traditional chain after $\mathcal{C} \cdot N$ confirmations. That is:

$$P'(q; c = \mathcal{C}N; N_1 = 1) \approx P'(q; c = \mathcal{C}; N_1 = N) \tag{32}$$

In fact, we can take this further. The relationship that exists is not only at the extremes, but between them too[91] — it must be if confirmations are equivalent. This is the *extended* CEC:

$$\begin{aligned} \forall a \in \{1,\dots,N\} : P'\left(q; c = \frac{\mathcal{C}N}{a}; N_1 = a\right) &\text{ is approximately constant} \\ \implies P'\left(q; c = \frac{\mathcal{C}N}{a}; N_1 = a\right) &\approx P(q; c = \mathcal{C}N) \end{aligned} \tag{33}$$

[90] Analysis of hashrate-based double-spending (Meni Rosenfeld; 2012)

[91] Since $\mathcal{C}N/a$, where a is the number of simplex-chains, in Equation 33 is not necessarily an integer, we'll assume there exists a valid generalization of Rosenfeld's solution for $c \in \mathbb{R}_{>0}$.

Notice that we don't need to know a closed form of $P'(q, c, N_1)$ to test the CEC: all cases *that we can measure* are equivalent to some *other, traditional* case, and we can convert between them.[92] This is the basis of the method used to test this hypothesis.

If Equation 33 is correct, it indicates that there exists a *generalization* of traditional blockchains (which have $N_1 = 1$); particularly, that there is *another dimension* (N_1; the size of the simplex) which is geometrically related to the first dimension, c.

Note: this experiment does *not* test the conversion of chain-work — the same hashing algorithm is used for each simplex-chain and no conversion of work takes place.

4.10.4.2 Method

I have implemented a simulation of a blockchain network that is suitable for this experiment. Particularly:

- The implementation is modular. It supports multiple: attack strategies; blockchain data structures; hashing algorithms (for PoW); and fork rules.
- An executable is produced that can run a single instance of a simulated attack, and takes a variety of parameters. (This is looped through via bash script to obtain CSVs of the results.)
- The implementation supports both traditional blockchains ($N_1 = 1$) and PoR/simplexes.
- It is fast.

The source code for this model blockchain network is contained in the `/experiments/por-sim-rs/` folder of this paper's repository.

While the simulation supports blockchains with single-parent blocks and a "longest chain" fork rule, these were not used in this experiment. Rather, multi-parent blocks were used (which simulates both GHOST and block-DAGs), along with the weight-based fork rule — these are required by UT.

Table 4: Table of parameters necessary to generate the main CEC experimental results. Only those parameters marked as variable were altered while generating the main results. Some parameters are omitted as they are not directly relevant to this experiment.

Parameter	Variable	Significance
q (ratio)	Yes	Attacker's proportion of the global hash-rate.
c (confirmations)	Yes	Work-equivalent of 'c local confirmations' waiting time.
N_1 (chains)	Yes	The number simplex-chains ($N_1 = 1$ for traditional blockchains).
${B_f}^{-1}$ (ticks)	No	Artificial units of time used by the DAA.
DAA_N (blocks)	No	The number of blocks over which the DAA operates.
H (hashes)	No	The network-wide average hashes per chain per tick.

When choosing values for ${B_f}^{-1}$ and H, we want them to be low enough for the simulation to be efficient, but large enough to ensure that there are no issues with limits-of-accuracy. To this end, both were chosen to be 75. This means that, prior to the doublespend attack beginning, the average difficulty is around 5625.

We also want to consider DAA_N — but there isn't an issue of *precision* in this case. The DAA used[93] adjusts every block, so lower values of DAA_N mean the DAA is more reactive to a drop

[92] This is not strictly true: Rosenfeld's solution requires an integer value for c. This is due to his solution assuming a static difficulty and the longest-chain fork rule; the *construction* is consistent with Bitcoin (provided the difficulty doesn't adjust), but not with chains that recalculate difficulty every block. Additionally, when using *work* instead of *height* as the measure of an attack's success, fractional values of c have real meaning — especially with a reactive DAA and $N_1 \gg 1$. Both factors turn out to be unproblematic for us, though. Rosenfeld's solution has reach beyond those assumptions and observed results converge with it for larger values of DAA_N.

[93] The simulation uses the algorithm named DAA-2 from An Economic Analysis of Difficulty Adjustment Algorithms in Proof-of-Work Blockchain Systems (Noda, Okumura, Hashimoto; 2020).

in hash-rate. If the DAA is more reactive, then attackers need to do *a little bit more work* since, on average, each of their blocks will weigh less than an honest block at an equivalent height. If the attacker's blocks (on average) weigh less, then the fork rule will favor an honest chain-segment (which has, on average, heavier blocks) and the attacker will need at least one *extra* block for their chain-segment be favored. The only way to overcome this, for shorter attacks or attacks against smaller simplexes, is for the attacker to be *a little bit luckier than usual.* (The effect is less significant with long attacks or attacks against larger simplexes since $P' \to 0$ anyway.)

Aside This explanation — that a smaller DAA_N means the attacker needs to be luckier — was not something I expected. It is somewhat speculative, and is included here to explain why smaller simplexes (3 to 10 chains) appear to have better security properties than the analytical solution predicts, and why we should prefer larger values of DAA_N if we want to match expected results.

In essence, a shorter DAA_N means that P' approaches zero faster. Since we want to *match* analytical results, we want a *larger* value of DAA_N for this experiment — this is analogous to an assumption of Rosenfeld's solution: *constant difficulty.*

So that we match the analytical solution's assumptions, a value of $\mathrm{DAA}_N = 500$ was chosen. The most important thing is that $\mathrm{DAA}_N \gg c$, particularly regarding traditional chains.[94]

Aside In practice, we should seriously consider smaller values of DAA_N for the security benefit.

With regards to the other values — q, c, and N_1 — they will be varied as part of the experiment so that we can generate each data-point. For each data-point, we'll gather at least 9,000 samples (this is indicated by "$n \geq 9000$" in these figures' legends, where n in this context is the number of samples).

The goal is to generate multiple data series that the CEC predicts to align. To that end, the functions in Table 5 are measured and graphed on the same axes. If the CEC is true then we expect: $f_1(x) \approx f_2(x) \approx f_3(x) \approx f_4(x) \approx f_5(x)$.

Table 5: Functions that will be simultaneously graphed to test the CEC. The main results are generated for $q \in \{0.4, 0.44\}$ and $C = 5$. The CEC predicts: $f_1(x) \approx f_2(x) \approx f_3(x) \approx f_4(x) \approx f_5(x)$.

Function	Description
$f_1(x) = P(q; c = Cx)$	Baseline: Rosenfeld's analytical solution (calculated)
$f_2(x) = P'(q; c = Cx; N_1 = 1)$	Simulated doublespend against a traditional chain
$f_3(x) = P'(q; c = C; N_1 = x)$	Simulated doublespend against a simplex
$f_4(x) = P'(q; c = 2C; N_1 = x/2)$	Simulated doublespend against a simplex (via CEC)
$f_5(x) = P'(q; c = 4C; N_1 = x/4)$	Simulated doublespend against a simplex (via CEC)

4.10.4.3 Error Correction Iteration

Following the initial implementation of PoR in the simulator, results were promising but did not align with the theoretical prediction (based on Rosenfeld's analytical derivation of $P(q; c)$). At lower values of c and/or N_1, the results suggested that the simulation was *more* secure than expected. At higher values of N_1, P' did not appear to approach 0, instead approaching $\sim$0.05. This hinted at the existence of potential errors in the simulator framework.

Given the nature of writing a simulation (compared to writing a production blockchain), many simplifications were made. Some of these simplifications shouldn't matter, but some might. How do

[94] The differential effect seems to diminish as an attack goes on, so DAA_N is particularly important for *lower* values of c. $\mathrm{DAA}_N \gg c$ matters when c is small, but not so much when c is large. In other words: if the doublespend occurs over 600 blocks, the value of DAA_N doesn't matter much; but if the doublespend occurs over 20 blocks, then the value of DAA_N is a significant variable.

we know if a given simplification (or implementation detail) introduced errors that were responsible for unexpected results?

In this case, sources of error were brainstormed, and evaluated in turn based on their potential to explain the largest sources of error. Of those, 4 major sources of error were corrected, which incrementally brought the simulation results closer and closer to the expected results (based on the CEC prediction). The error correction steps (including interim results) are briefly documented in Appendix C. Those corrections were:

1. Accounting for draft reflected work (explained in Section 4.10.3)
2. Randomizing hash-rates over the simplex ($p + q = 1$ is globally maintained)
3. Implementing the attacker's "Bonus Block"
4. Increasing the number of blocks over which the DAA operates ($\text{DAA}_N = 100 \rightarrow 500$)

Many possible sources of error were ruled out, too, like: choice of hashing algorithm (`xxh3`); values of ${B_f}^{-1}$, H, and other parameters; and implementation details like omitting certain validation steps.

Changing the simulation based on early results is dangerous though — why should we believe these results are valid? The answer is simple: all error corrections were motivated by *making the simulation more like a real blockchain*. No magic numbers were introduced, no values were scaled arbitrarily, simulations were not repeated to get better results, etc. Since we *know* that a simulation will naturally include simplifying assumptions (and omitted implementation details), we should expect that some of these simplifications might be significant. So focusing on, and correcting, those simplifications does not compromise the method. Rather, the fact that *these* corrections were the significant ones is consistent with the idea that *the simulation is consistent with real-world blockchain networks*; that the simulation *works*.

4.10.4.4 Validating the Model

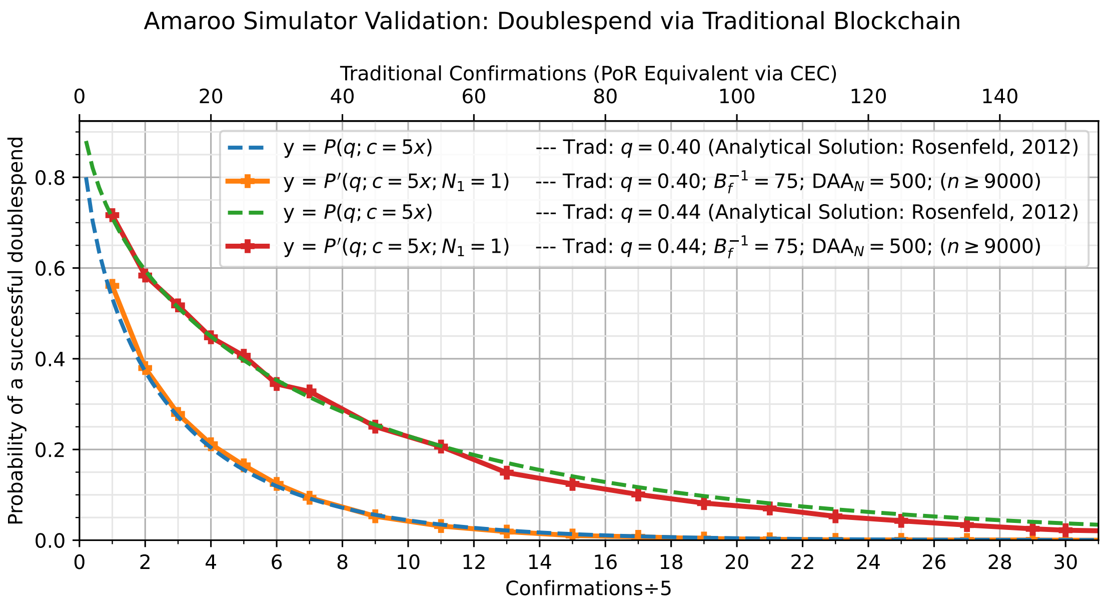


Figure 27: A comparison of the measured probability of a successful doublespend against a traditional blockchain compared with Rosenfeld's analytical solution, for $q \in \{0.4, 0.44\}$. That the measured results converge with the analytical solution is taken to be validation of the simulation — i.e., it *works* and is *useful* in the context of this experiment.

At a minimum, we should expect the simulation to faithfully replicate doublespends against *traditional* chains — i.e., $f_1(x) \approx f_2(x)$. Results of this, for $q \in \{0.4, 0.44\}$, are shown in Figure 27.

Figure 27 shows that the simulation *does* align with Rosenfeld's solution, especially at values of $q \lesssim 0.40$. It is consistent with Equation 31. As q increases, statistical artifacts and other inaccuracies *do* become more significant, but not enough to compromise the results — the largest differences (for $q = 0.44$ and $c = 5$) are around 50 parts per thousand (Figure 28) and typically less than this. There are more significant differences at $q = 0.48$,[95] however this is close to limiting threshold of blockchain security in general ($q < 0.5$), so we should expect larger error in the simulation results. Such differences are taken to be *non-critical*, i.e., they do not compromise the results.

4.10.4.5 Results

In this section we will only look at results for $q \in \{0.4, 0.44\}$, though results for $q = 0.48$ were also generated (among other combinations of parameters). Results that are omitted from this section are available in Appendix C.

First, do results of the simplest version of the CEC match predictions? Figure 28 graphs f_1, f_2, f_3 and the results are consistent with predictions (Equation 32).

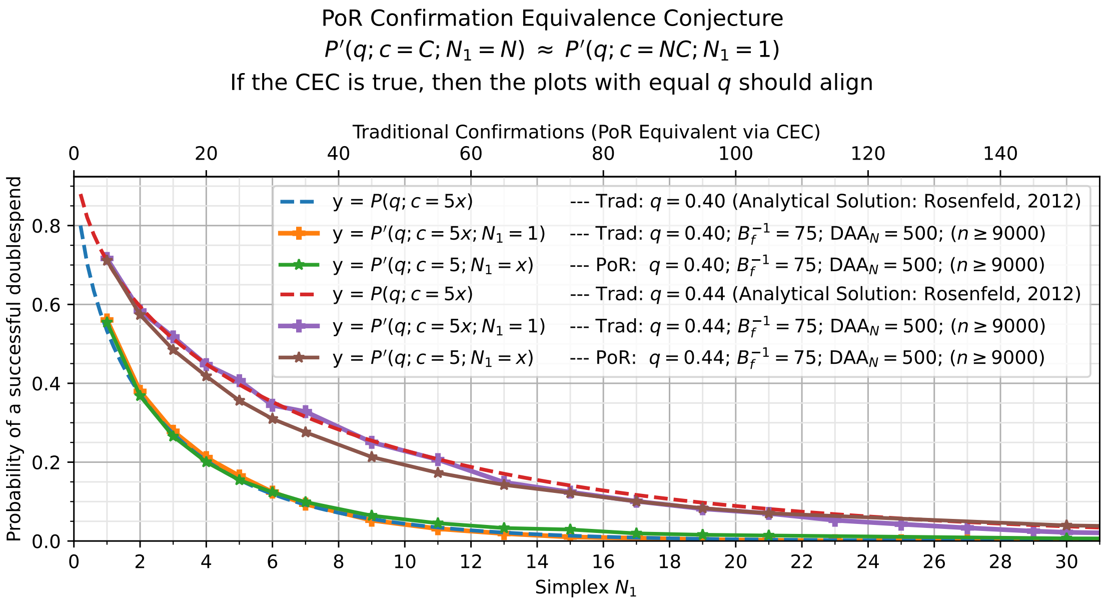


Figure 28: A comparison of the measured probabilities, for $q \in \{0.4, 0.44\}$, of a successful double-spend between: a traditional blockchain, a simplex, and Rosenfeld's analytical solution. That the measured results for a simplex (converted via the CEC) converge with the other results is taken to be evidence of $O(n)$ security and that the CEC is non-refuted.

While it appears that simplexes (for $q = 0.44, c = 5, N_1 \in [3, 10]$, at least) are slightly more secure than expected, this effect diminishes as N_1 increases. As we are about to see, it also diminishes as c increases. For this experiment, it is taken to be a *non-critical* error.

Second, what about the *general case?* Does the extended CEC hold true, also? Figure 29 graphs f_1, f_2, f_3, f_4, f_5 and these results are also consistent with predictions (Equation 33).

4.10.4.6 Conclusions

These results are, for the purpose of this paper, taken to indicate the following statements are true[96].

[95] See Appendix C.

[96] Experimental results never prove that something is correct, however, they are criticisms of *competing* explanations.

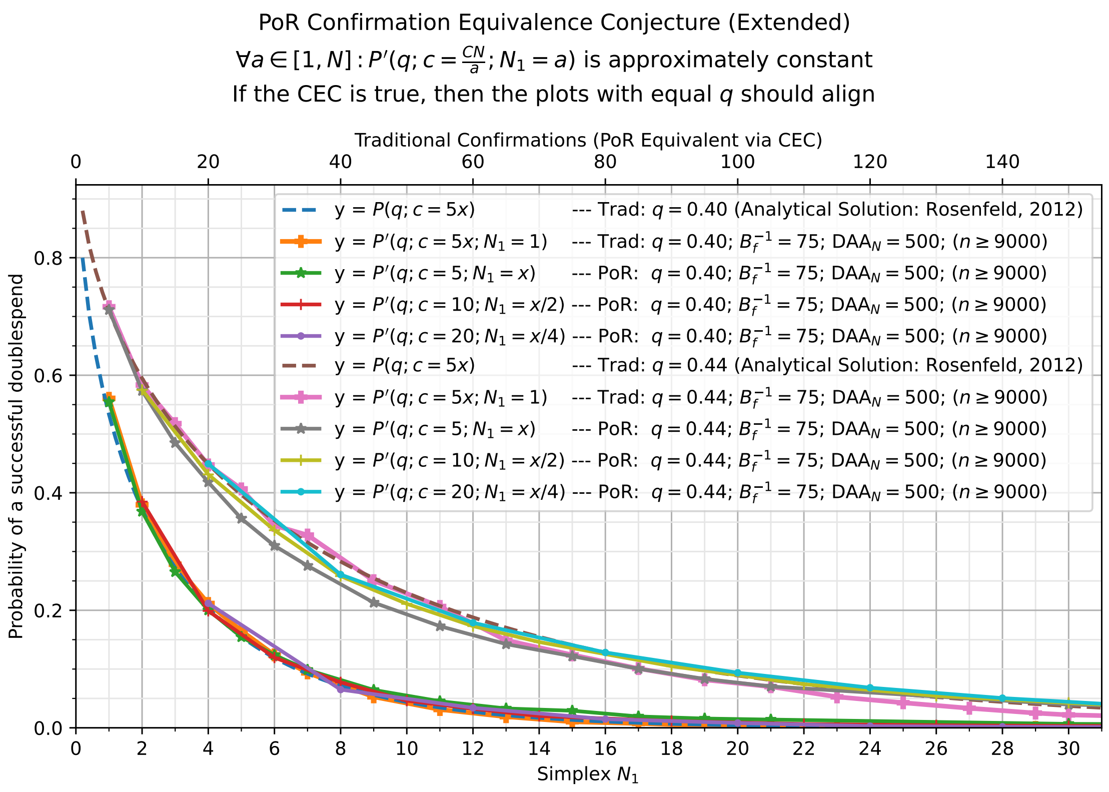


Figure 29: A comparison of the measured probabilities, for $q \in \{0.4, 0.44\}$, of a successful double-spend between: a traditional blockchain and multiple simplexes converted via the CEC. That the results converge as predicted is taken to mean that the extended CEC is non-refuted.

1. Sharing PoW security between blockchains without merged mining is possible.
2. PoR *works* as claimed when implemented correctly.
3. The *Confirmation Equivalence Conjecture* is, *generally and in principle*, true.
4. Simplexes are secure against a minority attacker and are thus $O(n)$ secure (assuming no relevant bottlenecks).

4.10.5 Closed Form of P'

Deriving a closed form of P' from first principles would be challenging. But perhaps we don't need to. First, let's consider the *PoR graph* without all the additional context of PoR and different simplex-chains — *what is it?* It is, more-or-less, a fairly-streamlined block-DAG. If we assume there is no latency, and thus there are no siblings or 'stale' blocks, then we expect a form similar to a traditional blockchain (with a very short block period). From this perspective, it is fairly obvious that we should expect $P \approx P'$ — they're trying to describe the *same thing.*

The block-DAG nature of the PoR graph may seem complicate the derivation, too, until we notice that comparisons between *linearized slices* of a block-DAG work exactly the same as comparisons

For example: we see that these experimental results *converge* as expected. This fact is NOT evidence that UT is secure, or that PoR works, or that the CEC is right. But, it is a *pre-emptive criticism* of ideas like: *The simulation does not work*, or *PoR is broken*, or *Simplexes aren't secure*, or *Sharing security between blockchains is impossible.* A decisive criticism of UT/PoR/CEC must therefore explain how these results are wrong — the explanation needs to include a *refutation* of these results. Such a refutation could be an explanation of how: the implementation was flawed, the methodology was flawed, or that UT cannot be effectively implemented as a production blockchain. No such explanations are currently known. In the *absence* of such criticisms, it is reasonable to work under the assumption that UT is secure, since *PoR works* and *the CEC is true* are currently (to my knowledge) the only viable explanation for these results.

between traditional chain-slices: *which is heavier?* Moreover, the *order* of blocks in the linearization *doesn't matter* since *all block-weight is counted* (and addition is commutative). Only the final comparison between the attacker's slice and the honest slice matters.

Therefore, we can use Equation 32 for the substitutions $c \to \mathcal{C}N$, then $P \to P'$, then $\mathcal{C} \to c; N \to N_1 \to n$ to arrive at a closed form. These substitutions are important because they *change the arguments* of P and P' without changing the algebraic description. Practically, in Equation 30, we are replacing c with cn.

$$\forall c, n \in \{1, 2, \dots\} : P'(q; c; n) = \begin{cases} 1 - \sum_{m=0}^{cn} \binom{m+cn-1}{m} (p^{cn} q^m - p^m q^{cn}), & \text{if } q < p \\ 1, & \text{otherwise} \end{cases} \tag{34}$$

Like Rosenfeld's derivation, this closed form comes with some assumptions. For example, we inherit the assumptions that the block period of all simplex-chains is identical and that the difficulty remains constant through the attack. Experimentally we found that randomizing the attacker's hash-rate over the simplex was required to match the results of the CEC, so this appears to be an assumption of this closed form, too.[97]

4.11 Intra-Simplex Cross-Chain Transactions

4.11.1 Introduction

Our goal in this section is to find a method for cross-chain transactions between any two simplex-chains that is safe, fast, succinct, and reliable.

- *Safe*: an adversary must perform a 51% attack to execute an invalid cross-chain transaction.
- *Fast*: $O(t) \leq O(\lambda) = O(1)$, where: t is the maximum delay between a cross-chain transaction being confirmed by a simplex-chain and the point at which the destination simplex-chain allows processing that transaction; λ is a security parameter (e.g., $\lambda = 4$ local confirmations).
- *Succinct*: for any cross-chain proof π, $O(\pi) < O(c)$ and $|\pi|$ is practical.
- *Reliable*: the above properties hold for all transactions in all simplex-chains.

Given everything we have discussed up to now, it's not clear how cross-chain transactions are possible. The problem is that miners on one simplex-chain do not validate the state transitions of other simplex-chains. Each simplex-chain, locally, has an $O(n/c)$ share of the network's security (assuming there are $O(c)$ many chains). An $O(n/c)$ adversary could theoretically produce *all* of the blocks for a single simplex-chain (e.g., by mining at a slight loss to push other miners out). The adversary could then create a very long chain of blocks that have valid headers, PoRs, etc, but also contain one or more fraudulent transactions. Perhaps the main chain eventually corrects to a non-fraudulent history (similar to Section 4.8.3) — but in that case we'd need to wait a *very* long time to be sure that such a correction *must* have happened. That violates the *fast* goal, so this eventual-correction idea fails to meet our overall goal.

Are we stuck? For a blockchain network to be scalable, it must be the case that miners are *not* required to validate more than one chain! Have we come all this way, attempting to deal with the core conflict of the trilemma, only to find that we just moved the conflict?

4.11.2 Why does SPV work for Bitcoin?

For the sake of simplicity, let's consider Bitcoin in isolation. Particularly, we'll assume that there are no other networks of similar size using the same PoW algorithm or compatible mining hardware.

A concise argument for the security of Bitcoin-SPV proofs is:

P.1 An SPV proof is a merkle branch proving that a transaction exists in a block.

P.2 All blocks are part of the same blockchain and history.

[97] Additionally, the simulation showed that attacks were less effective when the attacker's hash-rate was *uniformly* distributed over the simplex. See Section C.2.3 for more details.

P.3 All transactions that occur are in some block's merkle tree.

P.4 If a transaction is invalid, then the block is invalid.

P.5 If a block builds on an invalid block, then it is invalid.

P.6 If a miner produces an invalid block, they get no block reward.

P.7 $\implies$ Honest miners won't produce or build on invalid blocks.

P.8 $\implies$ For an adversary to produce an apparently valid (but fraudulent) proof leading to a *main chain* block, they'd need to out-compete honest miners (i.e., 51% attack the network).

P.9 $\implies$ Only valid transactions exist in the main chain.[98]

P.10 $\implies$ It's safe to accept SPV proofs for blocks in the main chain.

It appears that point 8 is our current sticking point for Simplex-SPV proofs.

But, we don't necessarily *need* to replicate point 8 if we could get to point 9 by other means.

4.11.3 Method

Note

The details below are an overview of Amaroo's cross-chain protocol. Full details comprise the Amaroo team's forthcoming paper on the subject.

4.11.3.1 Context

The purpose of UT's cross-chain protocol is to *maintain coherence* between simplex-chains. This means that all the essential properties we'd expect to hold for transactions on a single chain should hold between chains, too: coins are not created nor destroyed[99], coins can only be spent once, *the transaction executes entirely or not at all*, etc. There will obviously be some particular properties that are different, such as a delay between inputs being spent (on the sending chain) and outputs being created (on the receiving chain).

Before we address how cross-chain transactions will work, we first need to remember the context in which we're operating. Leading up to this point, we've constructed a network with some wildly different properties to traditional blockchains: ultra-fast confirmations, additional DoS and censorship resistance, shared security, higher order scaling, etc. These improvements are due to the specific combination of: *Proof of Reflection*, the use of chain-like block-DAGs in the PoR Graph and each simplex-chain, and the Axioms of Availability, Maximal Reflection, and Unified Ancestry. Each of these ideas, along with fraud proofs, will have a role to play in the cross-chain protocol described shortly.

Additionally, we are concerned about some possible attacks aimed at the cross-chain protocol. We are not necessarily concerned about how the protocol resolves during a 51% attack in this discussion since we already know that the network does not function in that context.

The first and most important thing to consider is *Under which circumstances can cross-chain transactions be invalidated?* If it is possible to invalidate a past cross-chain transaction, this could be used to perform a doublespend, or coin duplication, etc. Therefore, we should strive for a protocol that only permits such invalidation under a 51% attack (or worse). That way, we can be confident that the cross-chain protocol is not the weakest link in the chain, since local (intra-chain) transactions break down at that point, anyway.

[98]This isn't the full picture for some blockchains (e.g., Ethereum), even though SPV still works for them — a transaction that fails to execute can still be valid and produce a valid state transition. In these cases, an invalid state transition implies an invalid block, so it's customary to prove that some state has the right values rather than that a transaction exists.

[99]That isn't to say burning coins isn't possible, but it should never be an unintended byproduct of a cross-chain transaction.

The obvious way to invalidate a cross-chain transaction is to cause a reorganization on the origin chain. If the pivot of the reorganization is antecedent to the cross-chain transaction, then it is up to the attacker whether that transaction is included in the resulting canonical history. This outcome is self-evidently insecure. Therefore our question about cross-chain transaction invalidation has the same answer as questions like *Under which circumstances can a simplex-chain reorganization occur?* We are also concerned with the parameters around such a reorganization: *Is it dependent on the number of confirmations?*, *If so, how many?*, *What proportion of global hash-rate is required?*, etc.

Fortunately, we've done a lot of the legwork already to prove that the ordering of simplex-chain blocks is stable and works for cross-chain transactions. *Proof of Reflection* allows us to share security between simplex-chains. Block-DAGs and the PoR Graph provide some censorship resistance, and more importantly, are the basis for both the Axiom of Maximal Reflection and the Axiom of Unified Ancestry. These axioms go on to, respectively, bind the histories of each simplex-chain to one another and ensure that each chain's history spans all valid blocks of that chain in the PoR graph. The binding of histories is the basis for the *coherence* of the simplex, and PoR graph-spanning histories ensures that the ordering of blocks within each chain conforms to the ordering of blocks in the PoR graph. Therefore: to change the order of blocks within a simplex-chain requires that an attacker change the global ordering, which in turn requires a 51% attack. ■

Regarding network security, resisting this particular attack is paramount because this kind of reorganization *affects honest nodes.* By comparison, there are other attacks that do not affect honest nodes, but do affect, say, light clients.[100] These other attacks aren't as concerning provided that they only affect non-validating network participants. But! What happens when an invalid block is mined? It *will* be reflected, because miners of other simplex-chains do not validate the block. What we need is a method of *error correction* – one that is fast enough to ensure invalid blocks are dealt with before the cross-chain protocol allows those blocks to be processed. This role is filled by *fraud proofs.*

In UT, *fraud proofs* are automatically constructed by any full node during normal validation – the state transition function will return either a new state trie *or* a fraud proof. Once a fraud proof is constructed, it is broadcast similar to a transaction. Then, *any miner* from *any simplex-chain* can validate and include the fraud proof *as part of the PoR graph.* Due to Unified Ancestry, every simplex-chain calculates the expected chain-tips for every simplex-chain. When a fraud proof invalidates a block, the calculated chain-tips for a given simplex-chain are updated to exclude any invalid blocks from that chain's history. Thus, as long as fraud proofs are readily generated, broadcast, and included by other miners, the PoR graph knows which blocks are on-main valid blocks, and which are not.

There is, of course, some lag between an invalid block being mined, the fraud proof being generated, and the proof being included in the PoR graph. In a healthy network, the expected delay is $(N_1 B_f)^{-1} + o$ seconds, where o is a little overhead to account for the first node of that chain to receive and process the block. This is much less than a block period – even when a minority of miners are honest (particularly when $N_1 \gg p^{-1}$).

4.11.3.2 Construction

Typically, a cross-chain protocol will require two user interactions: one on the sending chain and one on the receiving chain. That is, there is some state modification on the first chain, and then, on the second, an acknowledgment of that and some associated action (like crediting an account). Based on the direction of information, order of actions, and the focus on those actions, we could describe these kinds of methods as *push-pull* methods. Information is 'pushed' from the sending chain, and, at some later point, 'pulled' into the receiving one – conceptually, at least. In practice the 'pull' requires the user to publish a proof about the sending chain to the receiving chain.

By contrast, a *push-push* method would be one where the receiving chain automatically processes the cross-chain output at the correct time. This means that no second action from the user is

[100]In this context light clients are not considered honest nodes (nor attacking nodes) since they do not contribute to global consensus.

required. Such a system must therefore *require* miners to process cross-chain transactions – but how do they know about transactions on other chains and whether they are valid? In our case, the Axiom of Availability means that all miners on all simplex-chains have all recent blocks from all other simplex-chains. Our use of fraud proofs means that all honest nodes quickly learn of invalid blocks. Thus, provided enough time has passed, we can be confident that miners know of all cross-chain transactions, and that they are, in fact, valid.

How long should we wait, and how do we measure how long we've waited? We need at least some delay to avoid a particularly lucky series of blocks from triggering cross-chain processing. Ideally, this processing should be as predictable as possible, and well distributed. We also want to avoid our delay method from being manipulated to cause earlier or later execution.

The delay duration itself is almost unconstrained: fraud proofs both propagate through the network well within one block period (at which point honest nodes are aware of them), and are included in the PoR graph within a block period, too. However, if we pick too short a period, then we risk some fragility in the network. We can technically recover if an invalid cross-chain transaction was processed before a corresponding fraud proof was recorded, but doing so robs honest miners of their block reward. So, whatever we pick, it needs to be something that is conservative enough for miners to be comfortable with.

Although it's somewhat arbitrary, 60 seconds (or 4 block periods at $B_f = 1/15$ Hz) seems like a reasonable choice. There is plenty of time for the fraud proof to propagate and be recorded, and the risk of an honest miner's block being invalidated is low.

How do we measure this duration, then? We could use block timestamps, although that may open us up to some kind of time-warp attack. We could use the number of confirmations on the sending or receiving chain, but there's local block variance to consider. We could use the change in volume of the simplex (i.e., the number of confirmations over all simplex chains since some block), but we don't have any guarantees about the progression of either the sending or receiving chain. Each of these has disadvantages, but if we take these breakpoints together, then we have a reliable single condition.

We can now summarize; when drafting an L block, the R blocks which are valid for cross-chain processing from the draft block's perspective are those which:

1. Have not been processed before; and
2. Have a timestamp at least λ block periods in the past; and
3. Have a volume at least $N_1\lambda$ blocks fewer than this block's volume; and
4. Are in the PoRs history of this block's λ-parent[101]; and
5. Are at least λ local confirmations deep.

These conditions are illustrated in Figure 30.

There is one slight oversight in this formulation that we need to fix. How do we know that the transactions in those blocks to be processed are all consistent? If we naively collect blocks according to the above conditions, then we don't – there might be conflicting transactions in off-main blocks. To correct this, we first use the four conditions to find the 1–4 most recent corresponding blocks (some conditions may resolve to the same block). Next, for chain R, we find the on-main MRCA (most recent common ancestor) of those blocks – this is the most recent block that will be processed for cross-chain transactions. We repeat this process for the parent of the draft block to find its corresponding MRCA by the same rules. Let's call these blocks R_α and R_β respectively. The set of blocks to process is: R_α and all on-main blocks in its history *minus* R_β and all on-main blocks in its history. This is the on-main "chain-slice" from R_α (inclusive) to R_β (exclusive). Since all of these blocks are on-main, any valid transactions from off-main blocks between R_α and R_β will be included in one of the on-main blocks. By inspection, we can observe that R_β for the current block was R_α for the parent block (or an earlier ancestor). By induction, this series of chain-slices continues all the way back to the genesis block (before which no chain-slices exist). Therefore,

[101]Definition: the 1-parent is the parent; the 2-parent is the parent of the 1-parent; the $(k+1)$-parent is the parent of the k-parent.

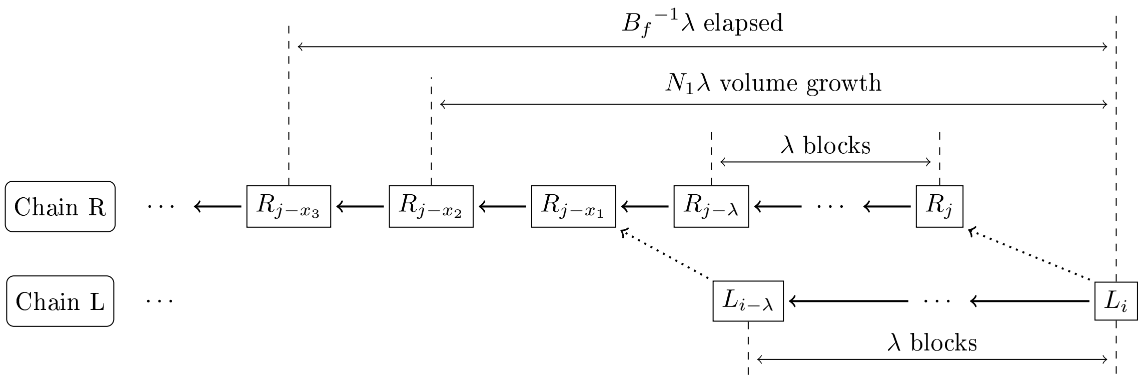


Figure 30: The conditions used to find R blocks that are valid for cross-chain transaction processing from the perspective of L_i. Particularly, we find up to 4 distinct blocks via 4 different paths. Note that these blocks can appear in any chronological order, and the specific blocks shown in the diagram are just one example of many possibilities. In this case, λ block periods ago resolves to R_{j-x_3}; volume at least $N_1\lambda$ blocks fewer resolves to R_{j-x_2}; the λ-parent of R_j resolves to $R_{j-\lambda}$; and the λ-parent of L_i resolves to $L_{i-\lambda}$.

every on-main block (and thus every valid transaction on that chain) is processed for cross-chain transactions at the earliest appropriate time.

The protocol for processing cross-chain transactions is now, simply: when drafting an L block, for each R chain, miners gather the chain-slice (of R blocks) that is valid for cross-chain processing, scan those blocks for incoming cross-chain outputs, and include those cross-chain outputs in the draft block.

Aside

In practice, these conditions aren't ideal because updating the reflections of a draft block may introduce new blocks which are valid for cross-chain processing, in turn requiring a complete recalculation of that block's output state. To avoid this, we replace item 4 and item 5 with similar conditions based on the draft block's parent and $\lambda - 1$. This keeps the set of blocks stable when updating the reflections of a draft block.

4.11.3.3 Evaluation

Safe Double-spending a cross-chain transaction requires a 51% attack on the entire network. Executing an invalid cross-chain transaction is not possible when there are honest nodes of that simplex-chain and honest miners of any simplex chain.

Fast Cross-chain transactions are processed based on conditions which are all $O(1)$ with respect to time, and therefore the expected delay in execution is $O(1)$.

Succinct No proof is explicitly recorded. If one did want to craft an explicit proof π, then it would be similar in size to a normal transaction proof (assuming headers from the surrounding PoR graph are available). Normal transaction proofs are $O(\log c)$, therefore, $O(\pi) = O(\log c) < O(c)$.

Reliable The consensus protocol requires that cross-chain transactions are processed if they are valid for cross-chain processing. Not doing so invalidates a block. Therefore, all cross-chain transactions on all simplex-chains will have these above properties.

Censorship Resistance This construction has a curious property. Say that an attacker, in an effort to perform an empty block DoS, is mining a simplex-chain at a loss, and has pushed out other miners. If you wish to make a local transaction, it could take much longer than a block period. An alternative is to make a cross-chain transaction to that simplex-chain from another simplex-chain instead. This will obviously take a minute or so to execute, but you will have a guarantee that it will be processed – the attacker cannot continue the attack without processing that cross-chain transaction. You could then 'rescue' your coins and send them to a more useful chain.[102] While this describes an unpleasant user experience, the takeaway is that it is yet another obstacle for an attacker to overcome.

4.11.4 Decoupled State Progression

The key to cross-chain transactions working is the decoupling of state progression combined with the property that cross-chain transactions can be read from R blocks without needing to validate the R chain – if it exists, then it is valid. Since part of this protocol involves asynchronous communication between chains, we need a method of error correction to ensure cross-chain transactions are safe: fraud proofs. Crucially, we expect that honest nodes (particularly those on other chains) are never fooled by an invalid block for long — much less than a block period. On this basis we can pick a safety parameter λ which (in combination with the conditions from earlier) provides enough buffer to ensure that the PoR graph remains coherent.

This decoupling is the essential component of UT that makes $O(c^2)$ scaling possible, and is a direct result of how we use PoR to create the PoR graph. The procedural elements are shown in Figure 31.

[102]This kind of operation would require some support from the transaction layer, e.g., a way to spend inputs indirectly. Such an output type would also need to contain instructions in case of a failure, since the sending chain cannot verify that the inputs to be spent (on the receiving chain) will be spendable when the cross-chain transaction is processed. Since we must scan blocks as part of the cross-chain processing, the branch taken should also be recorded in the block, alongside the corresponding output or transaction.

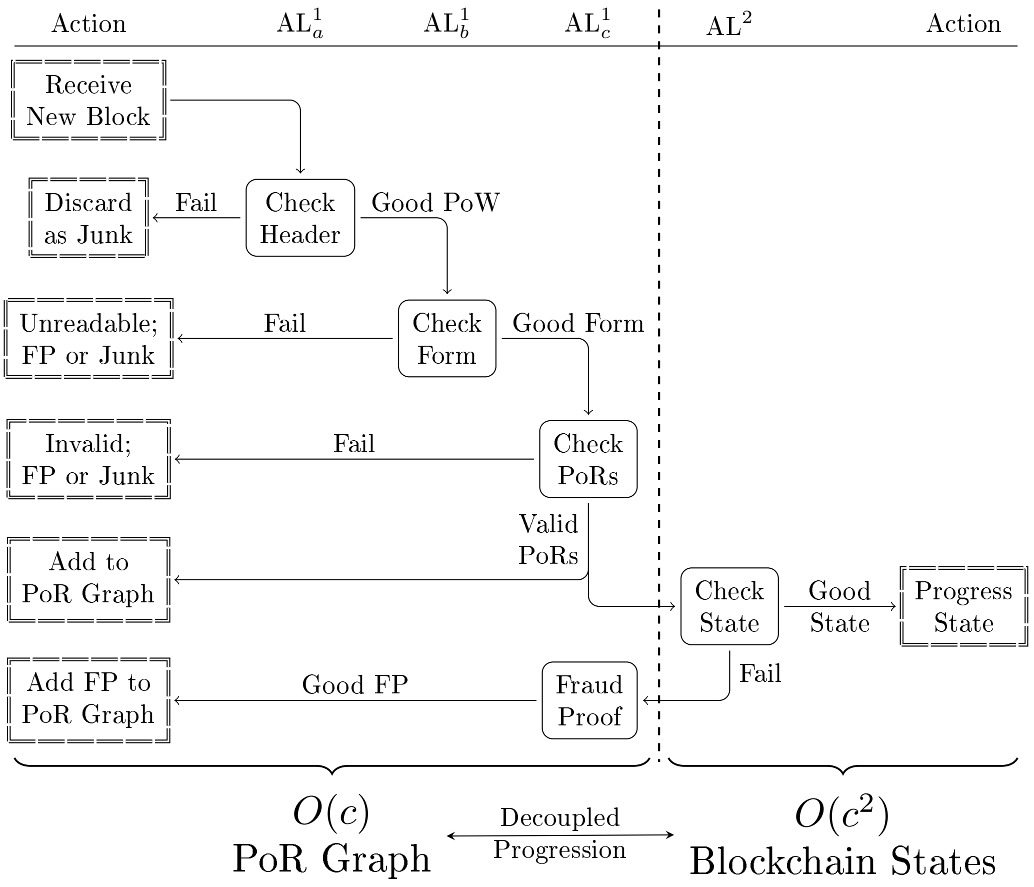


Figure 31: Flow chart of how the segmentation of state works in UT to achieve $O(c^2)$ capacity. "AL" is short for "Action Layer", a term invented here so that we can point to the various stages of PoR graph extension and state progression. AL^1 columns indicate actions that validating nodes of any simplex-chain perform on the PoR graph. The AL^2 column indicates that only validating nodes *of that chain* perform the associated action.

4.11.5 Fraud Proofs & Bribe Attacks: Derivations & Breakpoints

Let's assume that an attacker is willing to bribe miners to exclude a fraud proof. The goal of the attacker is to continue this attack beyond some breakpoint, so how much will that cost them over the course of the attack?

First, let's establish a baseline: what does a healthy simplex look like? *Healthy* meaning: no fraud proof is possible because all blocks and state-transitions are valid. We need to know what a healthy simplex (and its PoR graph) looks like because a successful attack will start off indistinguishable.

The defining quality of a healthy simplex is that additions to the PoR graph are highly interconnected. When new blocks are mined, they reflect all possible remote blocks and link back to all possible local blocks. Thus, there should be no partitions or asymmetric boundaries.

An *asymmetric boundary* emerges when a group of miners contributes *exclusively* to a *subgraph* of the full PoR graph (i.e., the subgraph that censors the fraud proof). The asymmetric boundary lies between the full (honest) PoR graph and the dishonest subgraph.

When an invalid block is mined, individual nodes of the network should rapidly become aware of the fraud proof — and we expect this if the P2P layer is working correctly. However, during an ongoing (successful) attack, that fraud proof is not recorded in the PoR graph. The fraud is 'invisible' when considering just the blockchain data itself, which means that SPV proofs and the like may be fraudulent.

When fraud is detected, the PoR graph should proceed through multiple phase changes until normality is restored (Figure 32).

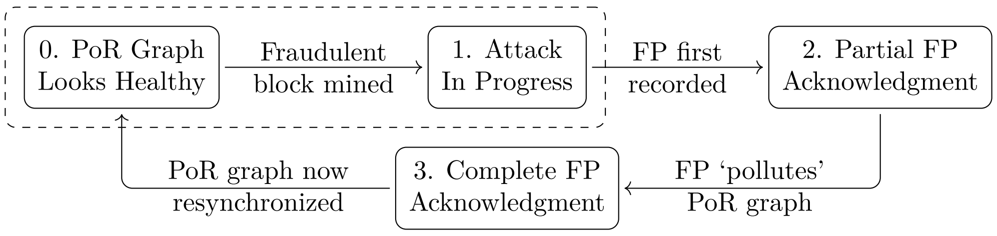

0. PoR Graph Looks Healthy
Fraudulent block mined
1. Attack In Progress
FP first recorded
2. Partial FP Acknowledgment
FP 'pollutes' PoR graph
3. Complete FP Acknowledgment
PoR graph now resynchronized

Figure 32: The phase transitions of a PoR graph when fraud occurs. The dashed outline around phase 0 and 1 indicates that it's not possible to observe this phase change on-chain. Nodes connected to the P2P network should be aware of the fraud, but no chain has yet included the fraud proof in the PoR graph.

An attacker always fails if the network reaches phase 3. The attacker's primary goal is to prevent the network transitioning from phase 1 to phase 2.

The dynamics of the attack change for both the attacker and each miner during each phase.

For example, there is a qualitative difference between phase 1 and phase 2, and it is in the attacker's interest to prevent the network transitioning to phase 2 if possible. Therefore it is most valuable for the attacker if they can prevent the FP being recorded. It is also more profitable for miners, since the size of bribes grow quadratically.

Once the network has reached phase 2, it is only a matter of time before the network reaches phase 3, at which point we are back at the start.

If the network is kept in phase 1, then the PoR graph and (nearly all) chain data is indistinguishable from a healthy simplex.

4.11.5.1 Phase 1 $\not\to$ Phase 2 on the Target Chain

The first honest block on the target chain must, by definition, link back to the fraudulent block as an invalid ancestor. Let's assume that there's a reward ($C_r H$) for publishing the fraud proof, where C_r is the block reward of the target chain, and H is the proportion of the block reward each miner gets. The additional reward for this block after k attacking blocks is $C_r Hk$. Thus, after H^{-1} blocks, the *effective* block reward for an honest miner will have doubled.

A new honest block thus becomes heavily incentivized over time.

To mitigate this, the attacker must raise the difficulty on the target chain such that all other potential miners find it comparatively unprofitable.

Compared to mining honestly, the attacker forgoes the reward of $C_r Hk$ for the k^{th} block. That reward is lost to excessive work securing the target chain which would otherwise earn the attacker that much revenue from publishing the fraud proof. Additionally, the attacker loses the block rewards each time — so these losses are $C_r(1 + Hk)$ per block after k blocks. Let $C_r \cdot l_m$ represent the total losses due to mining, and z the number of blocks for the attack: factoring out l_m in this way allows us to analyze the situation independent of the block reward. In order for the losses l_m to be greater than the rewards, we must have:

$$\begin{aligned} l_m &> \sum_{k=0}^{z} \Big(1 + Hk\Big) \\ &= z + 1 + H \sum_{k=0}^{z} k \\ &= z + 1 + H \frac{z(z+1)}{2} \end{aligned} \tag{35}$$

Additionally, the rise in difficulty of these blocks presents a practical limit for the attacker. When $k > qH^{-1}N_1$ the attacker is, *by definition*, unable to keep up the attack. That is because, at that point, the attacker's losses are $C_r(1 + H(qH^{-1}N_1)) = C_r(1 + qN_1)$. Assuming an even hash-rate distribution, $C_r N_1(p + q)$ represents the *sum* of *all* block rewards over the network. Therefore, the maximum output (measured in coins) of q proportion of the hash-rate, *by definition*, is $C_r N_1 q$. An attacker must have additional support to maintain this portion of the attack — when the mining losses per block (approx $C_r Hk$) exceed the attacker's mining capacity ($C_r N_1 q$).

$$\begin{aligned} C_r Hz &< C_r N_1 q \\ z &< qH^{-1}N_1 && H \geq {N_1}^{-1} \\ z &< q{N_1}^2 && \text{(worst case)} \end{aligned}$$

If $N_1 = 100$, then no attack is viable (without help) after 5000 blocks, unless $q > 0.5$. At that point, at least 50% of the network hash-rate needs to be dedicated to the target chain to make it unprofitable enough to dissuade honest miners.

4.11.5.2 Phase 1 $\not\to$ Phase 2 and $\mathbb{B}(z)$

To prevent the network transitioning from phase 1 to phase 2, the attacker incurs a cost. We can define this cost *in terms of the block reward.* Let the total cost to the attacker, in the k^{th} round, be $C_r Z_k$ coins. (Z_k is the cost in block rewards.)

- Each miner (on each simplex-chain) needs bribing (but only if they make a block).
- Each round, the attacker must pay $C_r Z_k$ in bribes (where Z_k is the bribe as a multiple of the block reward, and k is the round, starting at 1).
- These bribes are given to $(N_1 - 1)$ miners (we subtract 1 to exclude the target chain from this split). One bribe per R-chain.
- However, an attacker with q proportion of the hash-rate can recover $q(N_1 - 1)$ of these bribes, at most, and thus loses at least $p(N_1 - 1)$ of them.
- A miner receives $C_r Hk$ coins as a reward for publishing a fraud proof that invalidates k blocks.

- Each round, miners can choose to publish the fraud proof, which earns them $C_r Hk$ in rewards.
- Thus, each round, miners have an opportunity cost of $C_r Hk$ if they do *not* take the bribe.
- This only makes rational sense if the miner's bribe is greater than their opportunity cost: $C_r Z_k/(N_1 - 1) > C_r Hk$; *and* the miner is *guaranteed* to receive the bribe, regardless of whether the attack succeeds.
- The attack lasts z rounds.
- Let $C_r \cdot l_b$ be the total losses due to bribes.

Therefore, for the bribes of round k to be more than the rewards on all chains, we must have:

$$Z_k > kH(N_1 - 1)$$

Let $l_b = p(Z_1 + \cdots + Z_z)$ denote the expected cost of the attack for the z rounds. Substituting in the above:

$$\begin{aligned} l_b &> pH(N_1 - 1)\sum_{k=1}^{z} k \\ l_b &> pH(N_1 - 1)\frac{z(z+1)}{2} \end{aligned} \tag{36}$$

Let $\mathbb{B}(z)$ denote the minimum cost of the attack for z rounds, given our current understanding. Combining Equation 35 with Equation 36, and noting that $p + q = 1$ yields:

$$\mathbb{B}(z) = l_b + (p+q)l_m > H(N_1 p + q)\frac{z(z+1)}{2} + z + 1 \tag{37}$$

Observe that this is a lower bound on $\mathbb{B}(z)$. To reduce complexity, we'll also define $\mathbb{B}$ *in terms of the block reward* (i.e., in units of *block rewards* rather than *coins*). We can come back to it later if we need to.

$$\mathbb{B}(z) = \eta\frac{z(z+1)}{2} + z + 1 \qquad \text{where } \eta = H(N_1 p + q) \tag{38}$$

We should also note the inverse of $\mathbb{B}$, which gives us a lower bound on the number of blocks for which we should wait, given a total outgoing value, v (expressed as a multiple of the block reward).

$$\mathbb{B}^{-1}(v) = \frac{\sqrt{(\eta - 2)^2 + 8\eta v} - \eta - 2}{2\eta} \qquad \text{where } \eta = H(N_1 p + q) \tag{39}$$

4.11.5.3 Phase 1 → Phase 2

The network transitions from phase 1 to phase 2 as soon as a first miner produces a block that includes the fraud proof.

Once this happens, an asymmetry emerges: honest blocks (which acknowledge the fraud proof) produce proofs of reflection that are *not* usable by the dishonest miners. If the dishonest miners tried to use those PoRs, they'd produce invalid blocks unless they also acknowledge the fraud proof. Thus, using those PoRs effectively turns dishonest miners into honest miners.

4.11.5.4 Phase 2 → Phase 3

The duration of this transition depends on the proportion of honest miners: h. Let's define h via $h + d = 1$, where d is the proportion of dishonest miners, including the attacker.[103] Since d includes the attacker, we can say $d > q$, and $h < p$.

[103] Dishonest miners are those which will accept a bribe and participate in the fraud proof censorship attack, but are not necessarily controlled by the attacker directly. We assume that there are some miners that would accept a bribe for *this* attack, but not for *any* attack.

Similar to Section 4.8, the asymmetry between honest and dishonest miners means that, even for a small h, the chain-weight of the honest chain-tips will eventually dominate the chain-weight of dishonest chain-tips. After that point, each simplex-chain's main chain will include the fraud proof, and the dishonest chain-tips will never be able to out-weigh the honest ones.

Let us use a vector to represent the weight of the honest and dishonest chain segments (relative to their most recent common ancestor). This is purely for convenience so that we can analyze both at the same time. The top row is the weight of the honest chain segment, and the second row is the weight of the dishonest chain segment. At the start of the attack (t_0), neither chain segment has any blocks, so $t_0 = \vec{0}$.

Each round (block period), we expect that h proportion of new blocks are honest, and d proportion are dishonest. So, after 1 block period, we expect:

$$t_1 = t_0 + \begin{bmatrix} N_1 h \\ N_1 d \end{bmatrix} = N_1 \begin{bmatrix} h \\ d \end{bmatrix}$$

After 2 block periods, the h proportion of new blocks that are honest can use PoRs via the dishonest blocks. Due to the asymmetry, the dishonest blocks can not.

$$t_2 = t_1 + \begin{bmatrix} N_1(h+d) \\ N_1 d \end{bmatrix} = N_1 \begin{bmatrix} h \\ d \end{bmatrix} + N_1 \begin{bmatrix} 1 \\ d \end{bmatrix} = N_1 \begin{bmatrix} (h+1) \\ 2d \end{bmatrix}$$

Each round, the honest chain segments grow by the weight of N_1 blocks, but the dishonest chain segments grow by $N_1 d$ blocks. Therefore, at the k^{th} round, $k \geq 2$:

$$\begin{aligned}
t_k &= t_{k-1} + N_1 \begin{bmatrix} 1 \\ d \end{bmatrix} = t_{k-2} + 2N_1 \begin{bmatrix} 1 \\ d \end{bmatrix} \\
\text{observe: } t_k &= t_{k-i} + iN_1 \begin{bmatrix} 1 \\ d \end{bmatrix} \qquad \text{for some } i \in [1, k) \\
\text{set } i = k-1 \implies t_k &= t_1 + N_1(k-1) \begin{bmatrix} 1 \\ d \end{bmatrix} \\
t_k &= N_1 \begin{bmatrix} h \\ d \end{bmatrix} + N_1(k-1) \begin{bmatrix} 1 \\ d \end{bmatrix} \\
\therefore t_k &= N_1 \begin{bmatrix} h+k-1 \\ dk \end{bmatrix} \qquad (40)
\end{aligned}$$

Thus, the honest chain-tips dominate when

$$\begin{aligned}
h + k - 1 &> dk \\
k - dk &> 1 - h \\
k(1-d) &> 1 - h \\
hk &> 1 - h \\
k &> h^{-1} - 1
\end{aligned}$$

If $h = 1/N_1$, i.e., for each honest miner there are $(N_1 - 1)$ dishonest miners, then this inequality becomes: $k > N_1 - 1$.

If only 1% of miners refuse the bribe ($h = 0.01$), and that the other 99% of miners continue to mine only blocks that do not include the fraud proof, then we have: $k > 99$.

If we assume an honest minority of substantial size, $h = 1/3$, then $k > 2$. That is, the honest chain-tips are expected to dominate after 2 block periods if $1/3$ of miners are honest.

4.12 Expedited Transactions

In Section 4.7 we saw that UT's confirmation rate was $O(c)$, and that the expected confirmation *time* was $O(c^{-1})$. This is how long a user waits for a transaction to go from *newly* to *fully* confirmed.

In those calculations, we did *not* account for the time *between* the initial *broadcast* of a transaction and its *inclusion* in a block. To account for the duration of the whole the transaction's journey, we need to consider: broadcast latency (ϕ), the block period of the local chain, and the confirmation rate of the simplex as a whole.

$$\text{TimeToXConfirmations}(x) = \phi + \frac{1}{2B_f} + \frac{x-1}{N_1 B_f}$$

This process, end-to-end, is thus $O(\phi + 1 + c^{-1}) = O(1) > O(c^{-1})$. Can we do better?

Ideally, we'd want the *time to first confirmation* (TTFC) to be as short as possible — currently it is the bottleneck, after all. Naively, our first idea might be to use cross-chain transactions somehow: there are $\sim N_1$ chains that are potentially *sending* transactions, so that could provide the necessary coefficient to bring down the TTFC. This intuition does have some merit: we're going to need to use the *other chains* if we want a transaction to be 'confirmed' *during the period between blocks* of whichever chain the user is operating on. However, cross-chain transactions are way slower than we're seeking as they already wait for some number of confirmations.

We must also consider a *crucial difference* between local or cross-chain transactions and whatever solution we come up with: whether a transaction is spendable is only certain *in the context of a block.* Transactions without contexts are just some signed data. Even with a spendability proof, *that proof is against a pre-existing block.* Therefore, if there *were* some way to confirm a transaction after the last block and before the next, *we cannot tell if the transaction is valid or not.*

Thus, whatever the involvement of other simplex-chains is, those chains cannot know *for sure* that a foreign transaction is *guaranteed* to execute successfully. Requiring this would constitute *foreign validation* which would mean an $O(c^2)$ validation load, so we *cannot* have such a requirement. However, there is a corollary here: expedited transactions *do not depend* on the *validity* of foreign blocks and are thus not invalidated if a fraud proof is produced for the foreign block at some later point.

4.12.1 Protocol Design

This is enough to start sketching out expedited transactions.

1. R blocks must be able to include (specially flagged) L transactions.
2. L miners must now scan all *relevant* R blocks for expedited L transactions.
3. L miners must then *include and execute* those expedited transactions, and record whether each was successful or not. Duplicates are dropped.

This is a rudimentary protocol and we will need to improve it before it can be used. For example: what if an R miner includes a great many invalid L transactions? That would cause the L miner to waste valuable block space, reducing overall revenue.

DoS and Griefing We will prevent any significant impact *to miners* with an architecture that limits intentional DoS or griefing attacks.

The architectural change is simple:

- A block is *invalid* if it includes more than one[104] expedited transaction for any foreign chain.

Additionally, we can note that to support expedited transactions, a miner must scan foreign blocks. We already do this for cross-chain transactions, and miners already have the blocks due to the Axiom of Availability. There is no need for miners to include the raw transaction *again* if it has already been recorded. Therefore, an L miner only needs to reference this transaction by hash

[104]We could set this limit higher, but let's stick with one for now.

(similar to off-main transactions). This creates *a new asymmetry* which protects miners from griefing attacks: the attacking miner must use more block space than the victim.

Incentives and Fees A different kind of griefing is financial and might even happen without any malice: low-fee transactions. Miners normally have control over which transactions they include in their blocks and from this we get a functional fee market. However, if an R miner can include an expedited low-fee L transaction, and especially if this happens on many R chains at once, then L miners are at the mercy of whatever transactions are chosen *for* them.

To resolve this, we will *require*:

- Expedited transactions must pay an *equal* fee to *both* miners.

Additionally, we should note that *the fee is dependent on the successful execution of the transaction.* The foreign miner is therefore incentivized to consider only transactions with *current* spendability proofs — the standard of proof is the most recent L block. (Since we are considering *expedited* transactions particularly, we tolerate more stringent requirements than we'd require for simply relaying transactions.)

Miscellaneous Since high-fee expedited transactions are likely, it's natural to ask whether miners of that chain can include them. There's no reason to forbid this, and it further increases that miner's rewards, so we will allow it.

If a chain is *really* popular, then it's conceivable that almost all of the transactions are expedited. If it's possible for a maximum block size to be exceeded with *too many* expedited transactions then we'd have a deadlock on that chain. Therefore, when blocks are over the block size limit we need a special rule to allow such blocks.

To mitigate these issues, we will modify the protocol:

- Blocks can include any number of expedited transactions as on-main for the local chain.
- Blocks are only invalidated by the block size limit if they include on-main transactions other than the coinbase.

Synchronization Caveats We should note that synchronization of a single chain must change a little to work with expedited transactions. Particularly, along with the block itself, we will need some auxiliary data (the expedited transactions that were recorded in other chains). In terms of complexity, this will not be a problematic increase: we have at most N_1 expedited transactions per block, and $O(N_1) = O(c)$. Practically, this can mean synchronizing L chain nodes must download up to $\sim 5\times$ as much for periods of high transaction volume (we'll derive this shortly). On the flip side, expedited transactions can be pruned from the R chain that they were included in, the verifying R node only needs to know enough to recalculate the block root correctly.

The L Miner's Perspective A good feature increases the value of the network and it is in the rational self-interest of network participants to adopt it. Therefore, we can judge this feature from a miner's perspective to make sure it is preferable.

Miner's are concerned about making and re-making blocks. They want to be able to create them regularly and keep them up to date with minimal computational effort. Since we require (for fraud proofs) that the *output state* from each transaction is recorded in the block, any insertion of a transaction means re-executing transactions after the point of insertion to calculate new state roots. For simple transactions this is not too bad, but more complex transactions might have non-negligible overhead.

In general, we'd like the frequency of changes to be lowest for the first transactions and increase as we go. Ideally the way state is calculated should account for this so that miners don't unnecessarily have to recalculate state.

In terms of the transactions themselves, the cost of including an expedited transaction is two hashes: the transaction ID and the resulting state root. Since transactions are typically 250 bytes or more, we can estimate a 5× reduction in block size used for a similar transaction fee. Although new expedited transactions can arrive with any foreign block (which happens frequently), not every foreign block will have one, and they won't change order (or at least not often) when subsequent new ones arrive. That sounds good provided that the overhead is manageable.

Additionally, having the option to include expedited transactions for foreign chains is welcome. They'll have a decent fee and don't require recalculating any state. That said, a miner might miss out on some fees if they include an expedited transaction that some other miner already included in a simultaneous block. Choosing semi-randomly might help avoid collisions, but in the meantime the severity of this and efficacy of mitigation remains to be seen.

Overall, this is *win–win* provided that expedited transactions are not too complex, the overhead of recalculating state is not too high, and an intelligent strategy can be found for which foreign expedited transactions to include.

4.12.2 Security of Confirmations

Expedited transactions now have multiple stages of confirmation: first on the foreign chain, then on the local chain when they are actually executed. The first stage guarantees that the transaction will be executed, but not that it will succeed. Even a spendability proof against the prior L block is not sufficient since expedited transactions consume inputs, and we don't know which transactions will be processed *before* the expedited transaction is executed.

Aside This kind of confirmation is not unprecedented; off-main transactions are in a similar superposition-like state before they are executed in an on-main block.

Thus, we are left with the question: *Are these first-stage confirmations equivalent to second-stage confirmations?* One test is against double-spends: if they are easier to double-spend, then they are not equivalent.

Double-Spend Resistance If the attacker controls the inputs, then an attack depends on their ability to execute a conflicting transaction before the one in question. Therefore, *if* expedited transactions are *simultaneous or after on-main* transactions they would be vulnerable. However, if expedited transactions are *before* on-main transactions, then the inputs would have to be spent another way. Off-main transactions are executed before on-main ones, but off-main parents are always near-simultaneous with their siblings (due to the Axiom of Unified Ancestry) — we can therefore trivially calculate any conflicts.[105] Cross-chain transactions for a block are determined well in advance due to the delays built into the protocol, so any conflict here will be detectable and we can predict the outcome.[106] Other expedited transactions might conflict too, but we already know which those are (or will in short order), so again we can predict the outcome. There is one other candidate that we haven't thought about *at all* so far: some kind of automated transaction that is executed at the start of a block. However, since this would be entirely determined by the prior block, any outputs removed at this stage are already well known. These are all of the cases for transactions that may execute prior to the expedited transaction, and none allow for an advantage to the attacker provided we are watching for it. Therefore, expedited transactions are not vulnerable to double-spends, *provided* that they are executed *before* on-main transactions.

[105] An attacker might create a conflicting off-main block after the expedited transaction is first confirmed. This could spend the expedited transaction's inputs; to mitigate this vector, we require that expedited transactions are executed *before* any off-main transactions.

[106] Cross-chain transactions are technically outputs rather than transactions, so they never fail or conflict. However, it would be useful to send cross-chain transactions that can *try* to spend some inputs to new outputs. To ensure that such an output always executes successfully, such an output must include another output that is used if any of the inputs are not present or are invalid. The behavior of this output type, in english, is: try to spend the given inputs using these signatures; if this succeeds, insert the first set of outputs; otherwise, insert the second set of outputs. Care should be taken to ensure conservation of tokens depending on the branch taken.

Aside As sometimes happens when design work and development are simultaneous, some of the planning document may be written with the hindsight of solutions already implemented. The idea of automatic *system* transactions is an example — this has already been implemented in the Amaroo client. That's why I know they are deterministic and depend only on the parent block. The reason that they are at the start of a block is that they might contain heavy state operations that are inconvenient to do at the end. Having them at the start gives the same *result* as if they were at the end of the prior block, but, this way, the associated state calculations only ever have to be done once.

When Confirmations Differ The primary difference between the first and second stage of confirmation is that the first stage is *not* guaranteed to succeed.

Therefore, if we are to be *absolutely certain* that a transaction will succeed, then we *should* discriminate. Particularly, we need to wait for a local confirmation to actually process the block at minimum. At that point, however, we can *in hindsight* treat the confirmations as equivalent since they are double-spend resistant.

If the transaction is less significant (e.g., paying for groceries) then we don't need to worry too much. Such transactions are more confirmed than zero-confirmation transactions, and less than it will be after the next block.

Light-clients are a special case: if a local node is not available to validate blocks, then it is possible that we have an invalid block somewhere and we don't yet know about the fraud proof. In these cases, we need to wait as long as the cross-chain protocol waits to ensure that no fraud proof exists — about 60 seconds in practice. After this, the confirmations of both stages can be treated as normal.

4.12.2.1 Transaction Execution Order

For clarity, here is the execution order of transactions in a block:

1. Automated System Transactions (deterministic with on-main parent)
2. Cross-chain Transactions
3. Expedited Transactions
4. Off-main Transactions
5. On-main Transactions

4.12.3 $\mathbb{C}'$ and Time to First Confirmation

We can now calculate the time to first confirmation (TTFC) for expedited transactions.

The TTFC is the time it takes for an expedited transaction (with an appropriate fee) to be broadcast and confirmed on any foreign chain (of which there are N_1).

$$\text{TTFC} = \phi + \frac{1}{N_1 B_f}$$
$$O(\text{TTFC}) = O(c^{-1}) + O(c^{-1}) = O(c^{-1})$$

For a 150-simplex with 15 second block periods, TTFC $\approx$ 100 ms. This is on the order of ϕ and about as close to 'instant' as we can get.

We can now see that $\mathbb{C}'$ is accurate *from the moment an expedited transaction is broadcast.* For normal on-main transactions, $\mathbb{C}'$ is accurate *after* the first confirmation.

Additionally, for expedited transactions:

$$\text{TimeToXConfirmations}(x) = \phi + \frac{x}{N_1 B_f}$$

4.12.4 Effect on Chain-Capacity

There is a *significant* effect of expedited transactions that must be explored: the *effective capacity* of a chain is increased with demand, and the *overall* network capacity is slightly decreased.

Let us consider an exceptionally popular chain, L, with a growing demand for transactions.

To start with, the chain is at 100% utilization and fees are roughly equivalent to any other chain. As fee pressure increases, the first breakpoint for expedited transactions will be hit: the fee is more than $2\times$ the lowest fee of any other chain. At this point, it is profitable for that R miner to include the expedited transaction over local transactions. Additionally, transactions with this fee or higher should be counted as taking up about $1/5$ of the local block space that they normally would. With regards to capacity and utilization, we have traded one local transaction for five expedited transactions.

The next major breakpoint to consider is when the average fee passes $2\times$ the average lowest fee over *all* other chains. When this happens, the miners of half of all simplex-chains can increase their total fee intake by including expedited L transactions. On the (perhaps arbitrary) assumption that 50% of all L block space is used by expedited transactions, we have traded 50% of our naive capacity (for normal on-main transactions) for an equivalent 250% of our naive capacity in the form of expedited transactions. The effective capacity is therefore 300% of the naive capacity.

As the fee pressure increases further, other chains will become saturated with expedited transactions and there might be no space left for normal on-main transactions. This will barely register for other chains since only one expedited L transaction can be included in any R block. Let's also assume there are no duplicate expedited transactions. Since, under these conditions, we expect $\sim N_1$ expedited transactions per L block, we can use Equation 48 to say:

$$\text{ExpTxsPerBlock}_{\text{max}} = N_1 = \frac{k_1}{2B_f B_h} \tag{41}$$

For our typical lower-end parameters ($k_1 = 3000, B_f = 1/15, B_h = 112$): $\text{ExpTxsPerBlock}_{\text{max}} \approx 200$. By comparison, a block would typically contain about $k_1(2B_f \cdot \text{Tx}_{\text{avg}})^{-1} \approx 90$ transactions. Under these parameters, we can't even use all of the available block space! The 200 expedited transactions consume the space of $\sim$40 normal transactions, so overall we could expect around 250 transactions per block at most (80% of which would be expedited).

This is a little bit of a surprise (that the maximum increase is so low), but if we look to Equation 41 we can observe a symmetry with the formula for naive capacity:

$$\text{NormTxsPerBlock}_{\text{max}} = \frac{k_1}{2B_f \cdot \text{Tx}_{\text{avg}}}$$

In effect, we are capped based on the ratio of Tx_{avg} to B_h. One consequence is that, at least in some cases, we can probably relax the one-per-chain requirement.

Alternate case: $\text{UT}_{1+\text{HOT}}$ What if B_h were much smaller, though? If we *effectively* have $B_h \le g$ (the size of a hash), then we have:

$$\text{ExpTxsPerBlock}_{\text{max}} \ge \frac{k_1}{2B_f g} \approx 700$$

Aside Ahh, that's more like it.

This will use the equivalent block space of approximately 140 normal transactions – more than we could otherwise handle. This will increase the block size beyond its normal limit, but not by that much. In this case, $140/90 \approx 1.56$, and since this is only considering the transaction-half of the block, the complete block is only $\sim 1.28\times$ larger than normal.

Even in the most extreme case, this is clearly still within $O(c)$ block-size limits. That said, a *synchronizing* node will need to download around $(700+140+90)/(90\times2) \approx 5\times$ more data to synchronize the chain over such periods. If we are in a situation more like this than the previous one, we should keep the one-per-chain requirement.

Aside In practice, these formulas are a rough estimate and should be taken as such. The main use of this analysis is to set ballpark expectations and plan accordingly. Real-world figures will be heavily dependent on a variety of implementation details.

The take-away from all this is that expedited transactions provide a *capacity buffer* for a chain when demand is high, and have the effect of *distributing fee expenditure and transaction load* over the simplex.

4.13 Initial Configuration

The current plan for the initial configuration of UT and the Amaroo network:

- **Block Size:** $k_1 \in [3000, 6000]$ B/s. An exact value will be chosen closer to genesis.
- **Block Frequencies:** $B_f = {}^1\!/_{15}$ Hz for all simplex-chains.
- **Block Rewards:** $B_r = {}^{\textsc{BaseReward}}\!/_{N_1}$ coins. That is: according to the issuance schedule (which determines BaseReward), each block (from any simplex-chain) provides a proportional share of the base reward.
- **Convertible Context:** Single Root Token (SRT; Section 2.4.1).
- **Difficulty Adjustment:** DAA-2 as in An Economic Analysis of Difficulty Adjustment Algorithms in Proof-of-Work Blockchain Systems (Noda, Okumura, Hashimoto; 2020), modified to work for block-DAGs. The exact window size (DAA_N) is TBD but $\text{DAA}_N = 100$ (∼25 minutes) satisfies our requirements. Section 4.10.4 has important contributions.
- **PoW Algorithms:** TBD, but the goals are for diversity in the design of mining rig hardware, avoiding imbalance due to the current distributions of mining hardware, attracting existing under-utilized hardware, and providing *multiple* different paths for technology development. Additionally, PoW algorithms *will* be used by multiple simplex-chains (miner resonance would not happen, otherwise). One of the PoW algorithms will be deliberately compatible with Bitcoin's double-SHA256 and Bitcoin ASICs.
- **UT_1 Consensus Primitives:** PoW + PoR; multiple hashing algorithms.
- **UT Variant:** Conservative version of +HOT. Capacity is somewhere between +HO and +HOT; the current truncation method is lossless.
- **UT_1 Transaction Capabilities:** UTXOs by default; secondary EUTXO and EVM/WASM subsystems. Special systems will allow for interoperability between subsystems.
- **Simplex-Chains:** Mostly homogenous in architecture, though each simplex-chain can differ in these ways: the PoW algorithm, which external (non-Amaroo) chains are imaged, and the specific integrations with those external chains.
- **Intra-Simplex Cross-Chain Security:** $\lambda = 4$, which corresponds to ∼60 seconds.
- **UT_2 and Dapp-Chains:** Limited support initially with a focus on PoA dapp-chains; secure and universal dapp-chains will require further development. Constrained by and dependent on further developments of suitable PoS + PoR consensus methods.
- **Simplex Size:** For reasons discussed in Section 6, the initial limit on the number of simplex-chains will be 50% of maximal N_1. This corresponds to ∼75% of maximal TPS capacity. The simplex will start with fewer chains at genesis, with more added later. Genesis estimate: between 15 and 30 simplex-chains (Σ TPS $\in [150, 700]$).

5 Scaling Complexity Analysis of *Ultra Terminum*

UT has two primary methods of scaling: horizontally via mutual PoR (simplex-chains), and vertically via one-way PoR (dapp-chains). Horizontal scaling via PoR is novel. Dapp-chains are similar to many of the sharding and pseudo-sharding ideas proposed for other networks (Polkadot, Eth2, etc), though there are fewer restrictions on dapp-chains in UT compared to other designs. Additionally, dapp-chains in UT are secured by the entire simplex. In the case of PoS dapp-chains, this provides *additional* security compared to 'naked' PoS chains. Hosting dapp-chains on many simplex-chains also provides greater system-wide maximum capacity than a network built upon a single base-chain.

A common method of sharding is to *nest* blockchains. For example, Ethereum 2 has *The Beacon Chain* — its root-chain (the single base-chain of a network).

Quote

> The Beacon Chain will conduct or coordinate the expanded network of shards and stakers. But it won't be like the Ethereum mainnet of today. It can't handle accounts or smart contracts.
>
> — *The Beacon Chain | ethereum.org (2021)*

This type of configuration, where a base-chain facilitates child-chains, is referred to as *nesting* in this section and in the context of UT's architecture and complexity. Base-chains are at the first level of nesting. The shards of Ethereum 2 are *a level of nesting* above the Beacon Chain, i.e., nesting level 2. UT's dapp-chains are also at nesting level 2.

Term

Base-chain: A chain that has no parent-chains; i.e., is at the base nesting level.

Sometimes (but not always) people use terms like *layer 2* to describe this sort of nesting, though such usage of *layer 2* is ambiguous and potentially misleading. It easily confuses nesting with off-chain scaling methods (such as payment channels, rollups, or ephemeral 'child' blockchains, e.g., Plasma), and it potentially misleads readers about the security properties of nested blockchains. Nested blockchains *can* faithfully inherit the security properties of their parent-chains, which is not the case for layer 2 solutions prior to finalization.

Furthermore, terms like *layer x* cannot accurately describe UT's design. Consider a PoS dapp-chain on UT. Would that dapp-chain be *layer 1* or *layer 2*? It would be misleading to call it *layer 2* whilst *directly comparable* chains (like Ethereum 2, Polkadot, or Cardano) are called *layer 1*. Such PoS UT dapp-chains have *all* the security qualities of an equivalent stand-alone PoS chains, *and more*. If they were called *layer 1* chains, then what is the simplex — *layer 0*? It is clear that the common idea behind *layer 1/2* scaling does not have sufficient capacity to accurately describe UT's simplex- and dapp-chains; it is inadequate.

5.1 Analysis Methodology

The following derivations focus on *throughput* of particular blockchain designs and scaling configurations. These derivations will let us evaluate the complexity of each design.

Raw throughput of a network, T_i, is measured in bytes/sec (B/s) for some level of nesting, i. T_i directly corresponds to a design's maximum transactions per second (TPS_i), where Tx_{avg} is the average size of a transaction, via:

$$\text{TPS}_i = {}^{T_i}/_{\text{Tx}_{\text{avg}}}$$

The raw B/s throughput of a chain at the i^{th} level of nesting is denoted by k_i. Note that T_i is a *calculated* value, but k_i is a *parameter* that may be chosen. An increase to k_i is effectively an increase in maximum block size.

We will also derive relationships between the maximum number of chains at a level of nesting, N_i, and the maximum network throughput at that level of nesting, T_i. For most existing blockchain designs, $N_1 = 1$.

With regards to simplexes, we are particularly concerned with the complexity of a *maximal* simplex — i.e., the simplex with the highest TPS possible.

Term **Maximal Simplex**: A simplex with the maximum TPS under given $O(c)$ constraints.

Additionally, $O(k_i)$ is *defined* as $O(k_i) \equiv O(c)$. This is reasonable provided there are no $O(c)$ bottlenecks, e.g., network bandwidth, CPU throughput, memory requirements, etc (Section 1.1.2).

5.2 Complexity of $O(c)$ Chains

Example: Bitcoin.

The *raw throughput*, k_1, can be calculated for existing chains (e.g., Bitcoin) via the product of the maximum block size, B_{max} (in bytes), and the block production frequency, B_f (in hertz, or s^{-1}):

$$k_1 = B_{\text{max}} \cdot B_f$$

The *throughput*, T_1, of an $O(c)$ chain is equivalent to its raw throughput:

$$T_1 = k_1$$

The complexity order of the network is given by $O(T_1) = O(k_1) = O(c)$ as expected.

Care should be taken to account for protocol extensions like *Segregated Witness* that effectively reduce the size of transactions (in SegWit's case, by $\sim 1/4$).

For Bitcoin — given $k_1 \approx 1700$ B/s, and transaction size $\text{Tx}_{\text{avg}} = 500 \cdot 3/4$ B — the maximum TPS is given by:

$$\text{TPS}_{\text{Bitcoin}} \approx \frac{1700}{\text{Tx}_{\text{avg}}} \approx 4.5$$

This is what we expect based on the measured real-world performance of Bitcoin.

5.3 Optimistic Complexity of $O(c^2)$ Chains

Examples: Ethereum 2, Polkadot.

Suppose the root-chain has a throughput of k_1 B/s and it can support up to N_2 nested chains. Those nested chains have headers of D_h bytes that are produced at a frequency of D_f (s^{-1}). If *all* headers of nested chains are recorded in the host chain, then each nested chain consumes *at least* $D_f \cdot D_h$ B/s of the root-chain's capacity.

Thus, N_2 is given by:

$$N_2 = \frac{k_1}{D_f \cdot D_h} \tag{42}$$

NB: For blockchains of this design: $N_1 = 1$.

If each nested chain has a throughput capacity of k_2 B/s, then:

$$\begin{aligned} T_2 &= \frac{k_1 \cdot k_2}{D_f \cdot D_h} \\ &\approx \frac{k^2}{D_f \cdot D_h} \end{aligned} \tag{43}$$

Thus $O(T_2) = O(c^2)$ as expected.

5.3.1 Effective Header Size

It's typical, though, that the headers of nested chains, alone, are not sufficient: additional data is required. When such data is required to be recorded on-chain (i.e., it cannot be deterministically regenerated), then the *effective* header size is the size of the raw header, plus the size of any auxiliary data.

Aside Note that, for the calculations in this paper, 'Ethereum 2' means the sharded beacon chain design as written in 2021, not the rollup-centric dankssharding approach being pursued as of January 2025.

For example, in an *Ethereum 2* beacon block, each shard has a header size of 280 B, but there is additional overhead. A reasonable lower-bound is that each header has an *effective* minimum header size of 312 B.[107]

In the case of *Polkadot*, it is measurable[108] that a typical minimum of 819 B is used in the `paraInclusion.candidateBacked` extrinsic (i.e., the transaction type that records parachain headers). So, a lower-bound on the effective header size of a parachain is 819 B (this does not include *bitfields*[109]).

In those situations, with regards to these capacity derivations, one can use the *effective* header size as a replacement for the *raw* header size.

5.4 Complexity of UT$_1$

There is no single root-chain for a collection of mutually reflecting blockchains (i.e., a simplex), so $N_1 \neq 1$. What is N_1 then? In a simplex, each chain has k_1 B/s capacity, but this is split between reflections and transactions. At this foundational level (where there is no nesting yet), headers are B_h bytes with a frequency of B_f Hz. There are N_1 simplex-chains.

For the purpose of Section 5 we will generally *not* consider the impact of *explicitly* including PoRs along with block headers (i.e., the +PoRs UT variants). The methods we use here are easily generalized to account for those variants, and associated analysis can be found in Section B.1. Unless otherwise stated, Section 5 analyzes the UT$_{+\text{OP}}$ variant.

Reflecting a single simplex-chain requires $B_f \cdot B_h$ B/s of capacity, and each simplex-chain must reflect $N_1 - 1 \approx N_1$ other simplex-chains. This means that a simplex-chain must reserve $N_1 \cdot B_f \cdot B_h$ B/s of its capacity for reflections, denoted by $k_{1,B} = N_1 \cdot B_f \cdot B_h$. Additionally, simplex-chains must reserve some capacity for transactions, $k_{1,tx}$.

Since simplex-chains must split their capacity between reflections and transactions, set:

$$\begin{aligned} k_1 &= k_{1,tx} + k_{1,B} \\ k_{1,tx} &= k_1 - k_{1,B} \end{aligned} \tag{44}$$

Given $k_{1,B} = N_1 \cdot B_f \cdot B_h$:

$$k_{1,tx} = k_1 - N_1 \cdot B_f \cdot B_h \tag{45}$$

[107]As of late September 2021, the Ethereum 2 sharding spec has capacity for 2:1 attestations to shards per block (with 64 shards), but only 32 B of each attestation is dedicated to sharding. The spec also has capacity for 4:1 shard headers to shards per block. It seems reasonable that capacity which exists will be used within reason. Thus a reasonable lower-bound for the effective header-size of shards is taken via: 1× headers per shard per block, 1× attestations per shard per block (which do not count towards effective header-size), and 1× 32 B per attestation per block. Shards have headers of 280 B, so the minimum effective header size is taken to be 312 B. (note: a required dependency of the referenced sharding spec is the October 2021 merge spec and October 2021 phase0 spec.)

[108]Whilst some parachain headers exist that are smaller than 819 B, it's not really significant for this analysis (a reduction of 10% wouldn't change much). We're already ignoring bitfields, and 819 B seems optimistic if we're interested in the *average* parachain header size, since many are larger. All-in-all, I guess that 819 B is a bit generous, and (ideally) all claims about existing chains in this paper err on the side of generosity.

[109]Bitfields is a Polkadot term — it's a list of hundreds of signatures, totalling > 14 kB per block on the current Kusama testnet (October 3rd 2021).

Since each simplex-chain reserves $k_{1,tx}$ B/s for transactions, the total throughput reserved for transactions will be $N_1 \cdot k_{1,tx}$. Thus:

$$\begin{aligned} T_1 &= N_1 \cdot k_{1,tx} && (46) \\ &= N_1(k_1 - N_1 \cdot B_f \cdot B_h) \\ &= N_1 \cdot k_1 - {N_1}^2 \cdot B_f \cdot B_h && (47) \end{aligned}$$

The optimal number of simplex-chains will maximize throughput. We can find that maxima via:

$$\frac{dT_1}{dN_1} = k_1 - 2 \cdot N_1 \cdot B_f \cdot B_h$$

At $\frac{dT_1}{dN_1} = 0$:

$$\begin{aligned} k_1 &= 2 \cdot N_1 \cdot B_f \cdot B_h \\ \therefore N_1 &= \frac{k_1}{2 \cdot B_f \cdot B_h} \end{aligned} \qquad (48)$$

Thus $O(N_1) = O(k_1) = O(c)$.

From Equation 47 and substituting N_1 from Equation 48:

$$\begin{aligned} T_1 &= k_1 \cdot \frac{k_1}{2 \cdot B_f \cdot B_h} - B_f \cdot B_h \cdot \frac{{k_1}^2}{4 \cdot {B_f}^2 \cdot {B_h}^2} \\ &= \frac{2{k_1}^2}{4 \cdot B_f \cdot B_h} - \frac{{k_1}^2}{4 \cdot B_f \cdot B_h} \\ &= \frac{{k_1}^2}{4 \cdot B_f \cdot B_h} \end{aligned} \qquad (49)$$

Thus $O(T_1) = O({k_1}^2) = O(c^2)$.

What are $k_{1,B}$ and $k_{1,tx}$ in terms of k_1? From Equation 46 and Equation 49:

$$N_1 \cdot k_{1,tx} = \frac{{k_1}^2}{4 \cdot B_f \cdot B_h}$$

Substituting N_1 from Equation 48 gives:

$$\begin{aligned} \frac{k_1 \cdot k_{1,tx}}{2 \cdot B_f \cdot B_h} &= \frac{{k_1}^2}{4 \cdot B_f \cdot B_h} \\ k_{1,tx} &= \frac{k_1}{2} \end{aligned}$$

Thus, from the definition of k_1 in Equation 44, we find that maximal throughput requires half of the block to be used for reflections:

$$k_{1,B} = \frac{k_1}{2}$$

5.5 Dapp-Chains and the Complexity of UT$_2$ and UT$_3$

5.5.1 Dapp-Chains

If a system supports nested chains, then we can say that for some throughput, T_i, at nesting level i, the $(i+1)^{th}$ nesting level can support N_{i+1} nested chains via:

$$N_{i+1} = \frac{T_i}{D_f \cdot D_h} \qquad (50)$$

Therefore, via the same logic used for Equation 43:

$$T_{i+1} = T_i \cdot \frac{k_{i+1}}{D_f \cdot D_h} \tag{51}$$

Note that this relationship only holds for the traditional sharding model of securing sharded chains via their inclusion in a parent-chain, e.g., UT's dapp-chains (UT$_2$) and dapp-dapp-chains (UT$_3$), or existing $O(c^2)$ designs.

Combining these yields:

$$N_{i+1} = \frac{T_{i+1}}{k_{i+1}} \tag{52}$$

5.5.2 UT with Dapp-Chains (UT$_2$)

Starting with Equation 49 and building on Equation 51:

$$\begin{aligned} T_1 &= \frac{{k_1}^2}{4 \cdot B_f \cdot B_h} \\ \therefore T_2 &= \frac{{k_1}^2 \cdot k_2}{4 \cdot B_f \cdot B_h \cdot D_f \cdot D_h} \end{aligned} \tag{53}$$

Thus $O(T_2) = O(c^3)$.

The maximum number of dapp-chains is given by:

$$\begin{aligned} N_2 &= \frac{T_2}{k_2} \\ &= \frac{{k_1}^2}{4 \cdot B_f \cdot B_h \cdot D_f \cdot D_h} \end{aligned}$$

When $D_h = B_h$ and $D_f = B_f$, note that $N_2 = {N_1}^2$.

5.5.3 UT with Dapp-Dapp-Chains (UT$_3$)

If we say each dapp-chain hosts shards or more dapp-chains (e.g., as a dapp-chain version of Eth2 or Polkadot would), then via Equation 51 and Equation 53,

$$\begin{aligned} T_3 &= \frac{T_2}{D_f \cdot D_h} \cdot k_3 \\ &= \frac{{k_1}^2 \cdot k_2 \cdot k_3}{4 \cdot B_f \cdot B_h \cdot {D_f}^2 \cdot {D_h}^2} \end{aligned} \tag{54}$$

Thus $O(T_3) = O(c^4)$.

Note that the derivation of T_3 presumes that the parameters D_f and D_h are the same for both dapp-chains and dapp-dapp-chains.

Via Equation 52 and Equation 54:

$$\begin{aligned} N_3 &= \frac{T_3}{k_3} \\ &= \frac{{k_1}^2 \cdot k_2}{4 \cdot B_f \cdot B_h \cdot {D_h}^2 \cdot {D_f}^2} \end{aligned}$$

5.6 Complexity of Cross-Chain SPV Proofs & Proofs of Reflection

5.6.1 Cross-Chain SPV Proofs

Each chain — at full capacity — operates with order $O(c)$ by definition. Thus its state has order $O(c)$ also. The size of SPV proofs scale logarithmically with the set you're proving membership of, e.g., the number of transactions, or size of the chain's state, etc. Thus, SPV proofs scale with order $O(\log_2 c)$.

For a given $O(c^j); j \in \{2, 3, 4\}$ configuration of UT (i.e., UT_1, UT_2, UT_3), a chain can process SPV proofs of state on another chain. For $j = 4$, the furthest that a transaction can occur from its host simplex-chain is in the 3rd level of nesting (i.e., a dapp-dapp-chain). It would require $j - 1$ SPV proofs to "ascend" from the host simplex-chain to a dapp-dapp-chain. However, given that full nodes of a dapp-dapp-chain are required to be full nodes of both the host dapp-chain and the host simplex-chain, transactions in that dapp-dapp-chain do not need to provide SPV proofs of state in either of those host chains — full nodes already have those details. That is: transactions which "descend" the levels of nesting can do so with $O(1)$ cost. SPV proofs are only required when transactions "ascend" the levels of nesting to other simplex-, dapp-, or dapp-dapp-chains.

Thus, the maximum number of SPV proofs required to prove state anywhere in a UT simplex is j.

Since j is constant, cross-chain SPV proofs therefore have order:

$$O(j \cdot \log_2 c) = O(\log_2 c) \tag{55}$$

5.6.2 Proofs of Reflection

A simplex-chain reflects $N_1 - 1 \approx N_1$ other simplex-chains. A merkle tree of reflected headers has order $O(N_1) = O(k_1) = O(c)$ and a corresponding proof size of order $O(\log_2 N_1) = O(\log_2 k_1) = O(\log_2 c)$. Since those other simplex-chains also have $\sim N_1$ reflections, proving reflection in those other $\sim N_1$ simplex-chains requires $\sim N_1$ merkle branches. Thus, the full set of reflection proofs, per simplex-chain, is $O(N_1 \cdot \log_2 N_1) = O(c \cdot \log_2 c)$.

Note: In a production system, these proofs can be excluded from blocks by treating them as droppable witnesses; see Section 4.2.

5.7 TPS Complexity Comparison

k: raw per-chain throughput (bytes/s)
B_f: simplex block frequency (s^{-1})
B_h: simplex block header size (bytes)
$D_f = B_f$: dapp-chain block frequency (s^{-1})
$D_h = B_h$: dapp-chain block header size (bytes)

NB: For the purposes of Table 6 and on, the average transaction size is taken to be 250 bytes.

Table 6: A comparison of the maximum theoretical transaction throughput (transactions per second; TPS) with parameters k, B_f, B_h for $O(c)$, Sharded $O(c^2)$ and the different $\text{UT}_{+\text{OP}}$ scaling configurations. Note that the *Sharded* $O(c^2)$ column is the theoretical optimal limit for sharded systems where all headers are recorded in the base-chain.

k, B_f, B_h	$O(c)$	Sharded $O(c^2)$	$\text{UT}_{1+\text{OP}}$ TPS	$\text{UT}_{2+\text{OP}}$ TPS	$\text{UT}_{3+\text{OP}}$ TPS
$3000, 1/15, 112$	12	4,821	1,205	484,295	1.95×10^8
$3000, 1/15, 84$	12	6,428	1,607	860,969	4.61×10^8
$3000, 1/30, 112$	12	9,642	2,410	1,937,181	1.56×10^9
$3000, 1/30, 84$	12	12,857	3,214	3,443,877	3.69×10^9
$3000, 1/60, 112$	12	19,285	4,821	7,748,724	1.25×10^{10}
$3000, 1/60, 84$	12	25,714	6,428	13,775,510	2.95×10^{10}

More detailed comparison tables can be found in Appendix B.

5.8 Bandwidth Complexity

5.8.1 Full Node

What data must a full node download? A full node must be able to *completely validate* a *single chain*. For a simplex-chain, provided that the PoRs and corresponding headers remain available (which they always do[110]), this means it must download: all blocks for that simplex-chain, and any auxiliary data necessary to verify PoRs. Note that bandwidth requirements differ based on which UT protocol variant is used.

For $\mathrm{UT}_{+\mathrm{OP}}$, the data a full node requires are: each block, the headers of all reflecting chains, and the missing branches for all PoRs. Network-wide, headers consume $N_1 \cdot B_f \cdot B_h$ B/s. For a single chain, PoRs use $N_1 \cdot B_f \cdot g \cdot \lceil \log_2 N_1 \rceil$ B/s of capacity, where g is the digest size of the hash used for merkle trees (usually 32 bytes). Let's denote the total bandwidth required for a full node Δs. In the worst case ($\mathrm{UT}_{+\mathrm{HO}}$ and $\mathrm{UT}_{+\mathrm{HOT}}$), where both headers and PoRs must be downloaded:

$$\begin{aligned}\Delta s &= k_1 + (N_1 \cdot B_f \cdot B_h) + (N_1 \cdot B_f \cdot g \cdot \lceil \log_2 N_1 \rceil) \\ &= k_1 + N_1 \cdot B_f \cdot (B_h + g \cdot \lceil \log_2 N_1 \rceil)\end{aligned}$$

Thus, with *omitted proofs*, $O(\Delta s) = O(k_1 + N_1 \cdot \log_2 N_1) = O(c \cdot \log_2 c)$.

However, with *explicit PoRs* (variants including +PoRs), $\Delta s \leq k_1 + N_1 \cdot B_f \cdot B_h = O(k_1) = O(c)$.

Aside The term $g \cdot \lceil \log_2 N_1 \rceil$ in these equations is the size of a merkle branch for a PoR. What if verkle trees are used instead? With a branching factor of 256, 32 byte commitments and proofs, and 1 byte location specifiers, this term should be replaced with $(1 + 32) \cdot \max(1, \log_{256} N_1)$. The complexity of these two terms is the same — $O(\log c)$ — but in practice verkle PoRs are less than half the size of merkle PoRs. For this reason, the numerical calculations used in the tables throughout this paper assume that the UT implementation uses verkle trees.

5.8.2 PoR Graph

We found the bandwidth requirements for fully reconstructing the PoR graph before in Section 4.6.1 involving the measure of propagation delay, ϕ seconds. Let's call the total bandwidth requirements for the PoR graph Δr and substitute in Equation 48.

$$\begin{aligned}\Delta r &= N_1 B_f (B_h + g(1 + N_1 B_f \phi)) \\ &= \frac{k_1}{2B_h}\Big(B_h + g\Big(1 + \frac{k_1 \phi}{2B_h}\Big)\Big) \\ &= \frac{k_1}{2}\Big(1 + \frac{g}{B_h} + \frac{g k_1 \phi}{2{B_h}^2}\Big) \qquad (56) \\ \implies O(\Delta r) &= O(c^2 \phi)\end{aligned}$$

As discussed in Section 4.6.1 it is unclear whether $O(c^2\phi) = O(c)$. However, we don't need to know $O(\Delta r)$ to estimate some real world values of Δr. Practically, Δr looks to be between k_1 and $3k_1$ for $k_1 = 3000$ — but, this is sensitive to ϕ.

If we try to restrict $\Delta r < k_1$, then we find that we require a large B_h or small k_1:

$$\begin{aligned}k_1 &> \frac{k_1}{2}\Big(1 + \frac{g}{B_h} + \frac{g k_1 \phi}{2{B_h}^2}\Big) \\ 2{B_h}^2 &> 2B_h g + g k_1 \phi\end{aligned}$$

[110] The necessary PoRs are, at the very least, part of other simplex-chains, so "always do" assumes that simplex-chains themselves remain available, excluding planned shutdown. Since all blockchain networks *depend* on the availability of their chains, this is a safe assumption.

$$k_1 < 2B_h(B_h - g)(g\phi)^{-1}$$

If $k_1 = 3000$, $\phi = 1$, we have $B_h \geq 235$ — much larger than what we've assumed to be the minimum so far. Achieving $\Delta r < 3000$ requires $\phi \lesssim 0.2$ given the usual parameters.

Regarding the terms from Equation 56 within the parentheses, it is clear that $1 + g/B_h$ will be a value around 1.3, give or take. What do we expect the value of $gk_1\phi/(2{B_h}^2)$ to be? If $B_h \in [50, 500]$, and $\phi \in [0.05, 2.0]$, then we can see that the coefficient of k_1 is, at most, 0.0128 (and $3.2 \cdot 10^{-7}$ at least). Thus, this last term dominates $(1 + g/B_h)$ when $gk_1\phi \gtrsim 2{B_h}^2$. To keep Δr practicable, we should therefore go to some effort to keep $gk_1\phi \lesssim 2{B_h}^2 z$ (where z is a constant near 1 that we can use to add some acceptable tolerance in lieu of improving ϕ).

Axiom of Unified Ancestry Optimization Using the Axiom of Unified Ancestry, we do not need to reference a block's parents directly, since they are implied by the PoR graph and the absence of fraud proofs. This results in a small optimization where the parent hash is replaced with the best tip of the longest PoR chain. Regarding Δr, $g(1 + N_1 B_f \phi) \to g N_1 B_f \phi$ and thus:

$$\begin{aligned}\Delta r &= N_1 B_f (B_h + g N_1 B_f \phi) \\ &= \frac{k_1}{2}\left(1 + \frac{gk_1\phi}{2{B_h}^2}\right)\end{aligned}$$

A maximal simplex of reasonable parameters has $\Delta r \approx$ 7,200 B/s. Restricting $\Delta r < k_1$ now implies $k_1 < 2{B_h}^2(g\phi)^{-1}$. If $k_1 = 3000$, $\phi = 1$, we have $B_h \geq 219$.

5.8.3 Complete Simplex

Aside This is also the bandwidth required by all active miners due to the Axiom of Availability. Miners do not verify the state transitions of each block, but do verify the integrity of the data structure and PoR graph (which is very cheap by comparison).

What about the bandwidth required to verify *the entire simplex*?

If miners temporarily keep the blocks of every simplex-chain (so that they can regenerate PoRs and verify that reflected headers correspond to existent blocks) then what is the complexity and burden of this?

The amount of network bandwidth, ΔS, required to download all blocks (as they are produced) across all simplex-chains is the overall transaction throughput, $(N_1 \cdot k_{1,tx})$, plus the data necessary to reconstruct the PoR graph: Δr. Any auxiliary data can be deterministically regenerated.

$$\begin{aligned}\Delta S &= N_1 \cdot k_{1,tx} + \Delta r \\ \implies O(\Delta S) &= O(c^2) + O(c^2\phi) = O(c^2)\end{aligned}$$

If this were more than just downloading all the blocks, then we'd do that instead:

$$\begin{aligned}\Delta S \leq N_1 \cdot k_1 &= \frac{{k_1}^2}{4 \cdot B_f \cdot B_h} \\ \implies O(\Delta S) &= O(c^2)\end{aligned}$$

It is clear that ΔS has order $O(c^2)$, but how bad is this? For $k_1 = 3000$, $B_f = 1/60$, and $B_h = 112$: $\Delta S \approx 1.2$ MB/s. With those figures: $N_1 \approx 800$ simplex-chains, $N_2 \approx 645{,}000$ dapp-chains, and maximum tps of $\sim 7.7 \times 10^6$ at nesting level 2. Decreasing block times to 15s correspondingly decreases the bandwidth requirements to 0.3 MB/s for a simplex with $\sim$200 chains, $\sim 40{,}000$ dapp-chains, and $\sim 484{,}000$ max tps.

While $O(c^2)$ bandwidth scaling is not ideal, it's clear that — especially in the early days of a UT simplex when there are fewer simplex-chains — there are tolerable configurations available; i.e., there is *excess capacity.*

Table 7: Chain-capacity and bandwidth requirements for $\text{UT}_{+\text{OP}}$: N_1, N_2, N_3, ΔS, Δr, and $\mathbb{C}'$ for various parameters.

k, B_f, B_h	N_1	N_2	N_3	ΔS (B/s)	Δr (B/s)	$\mathbb{C}'$ (Hz)
$3000, {}^1/_{15}, 112$	200	40,357	16,215,243	306,137	4,798.5	13.4
$3000, {}^1/_{15}, 84$	267	71,747	38,436,133	408,959	7,173.5	17.9
$3000, {}^1/_{30}, 112$	401	161,431	1.30×10^8	607,477	4,798.5	13.4
$3000, {}^1/_{30}, 84$	535	286,989	3.07×10^8	810,744	7,173.5	17.9
$3000, {}^1/_{60}, 112$	803	645,727	1.04×10^9	1,210,155	4,798.5	13.4
$3000, {}^1/_{60}, 84$	1,071	1,147,959	2.46×10^9	1,614,316	7,173.5	17.9

5.9 The Impact of Header Size

Equation 51 shows that UT's throughput is inversely proportional to the size of headers, D_h, for that given depth of nesting (this is true for UT_1, too). It also shows that throughput is inversely proportional to the block frequency, D_f, and proportional to chosen raw throughput, k.

Of these three values (header size, block frequency, and raw throughput), header size is the only value we *cannot* choose arbitrarily. To maintain overall throughput, doubling the header size requires one of: halving the block production frequency (i.e., doubling the block target time), doubling the chain's raw throughput, or some combination of those two options. One such combination would be to decrease the block production frequency by a factor of ${}^1/\sqrt{2}$ and increase the raw throughput by a factor of $\sqrt{2}$.

Changing all header sizes by some factor has different effects for different UT configurations. For UT_1, the effect on throughput is linearly proportional to the factor; doubling the header sizes reduces overall throughput by a factor of 2. However, for UT_2, the effect is quadratically proportional to the factor; doubling the header sizes will reduce overall throughput by a factor of 4! The relationship is even worse for UT_3, where the effect is cubicly proportional.

It is worth noting, though, that different header schemes can be used in each level of nesting. This means that if, say, dapp-chains need larger headers than simplex-chains, then there isn't a negative effect on the capacity of the simplex (i.e., the level(s) beneath).

This effect is not unique to UT, though. In general, any system of sharding is also affected in this manner when the headers of a child-chain are included in the parent-chain's blocks.

Practically, this effect means that a decrease to the size of headers has *increasing* marginal benefit. Compared to $O(c)$ blockchains (e.g., Bitcoin), efficient header schemes are far more important for UT and sharded blockchain networks.

5.10 Comparison of UT Variants

Table 8 and Table 10 show a comparison between UT variants. For comparisons over a range of parameters, see Appendix B.

Table 8: Comparison of UT variants' capacities with parameters: $k = 3000$ B/s; $B_f = 1/15$; $B_h = 84$ bytes; 250 byte transactions. "E. B_h" means the *effective* header-size. Simplex-headers *for the purposes of PoR* can shrink to less than the actual header-size due to the data that is excluded under the corresponding scaling configuration.

	N_1	Σ TPS$_1$	N_2	Σ TPS$_2$	$\mathbb{C}'$ (Hz)	E. B_h (B)	PoR (B)
UT$_{\text{+PoRs}}$	192	1,153	51,510	618,130	12.8	117	33
UT$_{\text{+PoRTs}}$	227	1,363	77,479	929,750	15.1	99	33
UT$_{\text{+HOPoRs}}$	273	1,940	86,648	1,039,779	18.2	67	35
UT$_{\text{+HOPoRTs}}$	326	2,319	131,797	1,581,574	21.7	56	40
UT$_{\text{+OP}}$	267	1,607	71,747	860,969	17.9	84	35
UT$_{\text{+OPT}}$	340	2,045	116,219	1,394,628	22.7	66	41
UT$_{\text{+HO}}$	703	4,218	188,337	2,260,044	46.9	32	54
UT$_{\text{+HOT}}$	1,406	8,437	479,403	5,752,840	93.8	16	60

Table 9: Comparison of UT variants' capacities; as in Table 8 with $k = 20000$ B/s.

	N_1	Σ TPS$_1$	N_2	Σ TPS$_2$	$\mathbb{C}'$ (Hz)	E. B_h (B)	PoR (B)
UT$_{\text{+PoRs}}$	1,023	42,256	1,886,448	1.51×10^8	68.2	142	58
UT$_{\text{+PoRTs}}$	1,168	48,009	2,727,810	2.18×10^8	77.9	125	59
UT$_{\text{+HOPoRs}}$	1,535	64,614	2,884,575	2.31×10^8	102.3	93	61
UT$_{\text{+HOPoRTs}}$	1,791	77,078	4,379,483	3.50×10^8	119.4	77	61
UT$_{\text{+OP}}$	1,785	71,428	3,188,775	2.55×10^8	119.0	84	61
UT$_{\text{+OPT}}$	2,272	90,909	5,165,289	4.13×10^8	151.5	66	62
UT$_{\text{+HO}}$	4,687	187,500	8,370,535	6.70×10^8	312.5	32	64
UT$_{\text{+HOT}}$	9,375	375,000	21,306,818	1.70×10^9	625.0	16	65

Table 10: Comparison of UT variants' network and storage requirements with parameters: $k = 3000$ B/s; $B_f = 1/15$; $B_h = 84$ bytes; 250 byte transactions. DTS$_{5yrs}$: The days to sync 5 years of a simplex-chain's history, including PoRs, with a fully utilized 10 MB/s network connection.

	Δs (B/s)	DTS$_{5yrs}$	Chain-GB/yr	Δr (B/s)	ΔS (B/s)	Σ DTS$_{5yrs}$
UT$_{\text{+PoRs}}$	3,000	0.55	94.7	4,106	292,567	53
UT$_{\text{+PoRTs}}$	3,000	0.55	94.7	3,073	343,981	62
UT$_{\text{+HOPoRs}}$	4,528	0.83	142.9	7,411	492,641	89
UT$_{\text{+HOPoRTs}}$	4,434	0.81	139.9	5,560	585,471	106
UT$_{\text{+OP}}$	3,617	0.66	114.2	7,173	408,959	74
UT$_{\text{+OPT}}$	3,939	0.72	124.3	5,995	517,359	94
UT$_{\text{+HO}}$	9,472	1.73	298.9	40,593	1,095,281	200
UT$_{\text{+HOT}}$	14,822	2.71	467.8	78,000	2,187,375	399

Table 11: Comparison of UT variants' network and storage requirements; as in Table 10 with $k = 20000$ B/s.

	Δs (B/s)	DTS$_{5yrs}$	Chain-GB/yr	Δr (B/s)	ΔS (B/s)	Σ DTS$_{5yrs}$
UT$_{\text{+PoRs}}$	20,000	3.65	631.2	82,331	10,646,442	1,944
UT$_{\text{+PoRTs}}$	20,000	3.65	631.2	54,890	12,057,258	2,201
UT$_{\text{+HOPoRs}}$	28,596	5.22	902.4	179,424	16,333,048	2,982
UT$_{\text{+HOPoRTs}}$	27,880	5.09	879.8	123,841	19,393,568	3,541
UT$_{\text{+OP}}$	27,307	4.99	861.7	240,566	18,097,709	3,305
UT$_{\text{+OPT}}$	29,454	5.38	929.5	196,078	22,923,351	4,186
UT$_{\text{+HO}}$	66,351	12.12	2,093.9	1,598,750	48,473,750	8,852
UT$_{\text{+HOT}}$	102,016	18.63	3,219.4	3,176,250	96,926,250	17,701

6 $\text{UT}_{\aleph}$: Tiling Simplexes

In a normal simplex, *all* simplex-chains mutually reflect *all other* simplex-chains. This method (UT_1) is useful, but, without dapp-chains, it is limited to $O(c^2)$ scalability.

Can we do better than $O(c^2)$ scalability, though? Is $\underline{O(n)}$ scalability possible?

When analyzing simplexes in Section 5, we saw that each chain in a simplex reserves $1/2$ of its capacity for reflections (with other simplex-chains) and $1/2$ of its capacity for transactions and dapp-chain headers. We could, however, use some of this *capacity for reflections* for a slightly different purpose. If we were to divide the *capacity for reflections* into multiple partitions, then we would have a *smaller simplex*, but could also use this new *excess capacity* for something else. Let's consider a simplex where $3/4$ of its capacity for reflections is reserved as *excess capacity*. Here, *internal reflections* are the mutual reflections between all simplex-chains that are part of the same simplex.

Simplex-chain Capacity

Internal Reflections	Excess Reflection Capacity	Transactions and Dapp-chain Headers
12.5%	37.5%	50%

What can we use that *excess capacity* for?

To answer this, we need to introduce a new *type* of simplex: a simplex *tile*. These are smaller simplexes that, unlike *maximal simplexes*, explicitly reserve some reflection capacity for the purpose of mutually reflecting simplex-chains *in other simplex-tiles*. Reflections with simplex-chains in other tiles are called *external reflections*.

Term

Simplex-Tile: A simplex which partitions its capacity for mutual reflections such that each simplex-chain mutually reflects all other simplex-chains in that tile, and all other simplex-chains in *neighboring* (or *adjacent*) tiles.

We can now replace the *Excess Reflection Capacity* in our simplex-chain capacity diagram like this:

Simplex-chain Capacity

Internal Reflections	External Reflections	External Reflections	External Reflections	Transactions and Dapp-chain Headers
12.5%	12.5%	12.5%	12.5%	50%

Simplex-tiles will be the foundational unit from which we construct *Simplex Tilings*.

6.1 Simplex Tilings

We connect simplex-tiles together, via mutual reflection, to create a simplex tiling. For the sake of brevity, when we say that a simplex-tile reflects another tile (or that a tile is *connected* to another tile), we mean that all simplex-chains in the first tile mutually reflect all simplex-chains in the second tile. A simplex tiling is thus a *graph* of interconnected tiles. Connected tiles are *neighbors* and are *adjacent* to one another in the tiling.

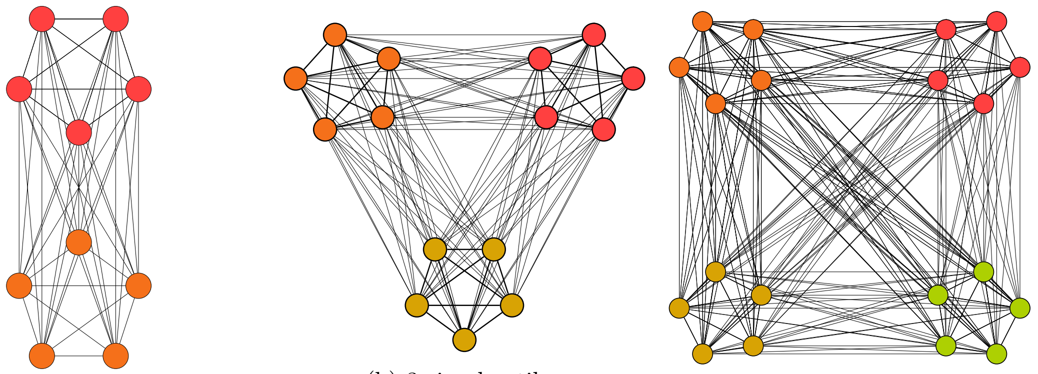
Term

Simplex Tiling: An interconnected graph of mutually reflecting simplex-tiles.

Before we look at some complex (and useful) tilings, let's consider three examples that are some of the simplest tilings possible: Figure 33. The examples contain 2, 3, and 4 simplex-tiles respectively.

Aside

Note: we are not yet concerned with the security properties of these examples — we need to build up to that first.



(a) 2 simplex-tiles

(b) 3 simplex-tiles

(c) 4 simplex-tiles

Figure 33: Trivial simplex tilings — these are all equivalent to a single simplex as each simplex-chain reflects all other simplex-chains in the network.

In those examples, notice that each simplex-tile reflects each other tile. This means that each simplex-chain is mutually reflecting *all* other simplex-chains (in all other tiles), so each of these tilings is equivalent to a standalone simplex. These tilings aren't particularly useful to us in practice, though, because we don't have a way to increase network-wide capacity beyond $O(c^2)$.

If we want $O(n)$ capacity, then we'll need to come up with a new pattern for how to connect tiles together and how to add new tiles. Specifically: the pattern must *always* have room for more simplex-tiles, and each tile added must, on average, add *more* than 1 new spot for additional tiles.[111]

6.2 Tile Valence

The *valence* (v) of a tile is the number of other tiles that it can connect to. In our example earlier, we split our simplex-tile's capacity for reflections into 4 equal parts — each part was 12.5% of each simplex-chain's *total* capacity. Since 1 of those parts is reserved for internal reflections, and 3 parts are reserved for external reflections, that simplex-tile has a valence of $v = 3$. If, instead, we split our simplex-tile's capacity for reflections into *5* equal parts, then each part would be 10% of each simplex-chain's *total* capacity, and the tile would have $v = 4$.

We don't *have* to use a valence of 3 — we can split up a tile's capacity for reflections any way that we like. Each tile could even have different valences, too, but this makes designing a tiling more complex. For the sake of simplicity, we will only consider tilings where each tile has the same capacity and the same valence. When we say that a *tiling* (rather than a tile) has a valence of v, we mean that *each tile in the tiling* has a valence of v.

If a tiling has a valence of $v = 0$, then all capacity for reflections is reserved for internal reflections — so this is just a normal simplex.

If a tiling has a valence of $v = 1$, then the tiling is limited to 2 tiles at most — there is only one solution. We end up with a tiling that's equivalent to a normal simplex, though, since all simplex-chains *must* reflect all other simplex-chains. The complete tiling with $v = 1$, which is the only possible tiling, is shown in Figure 33a.

If a tiling has a valence of $v = 2$, then there is a countably infinite number of solutions. The trivial solution (which has 3 tiles) is where each simplex-tile is connected to all other simplex-tiles — this

[111]There are tilings where a tile consumes more than one free spot. For those tilings, on average each tile must add more spots than it consumes.

is shown in Figure 33b. We can create a tiling of $v = 2$ with 4 tiles, too: $A \leftrightarrow B \leftrightarrow C \leftrightarrow D \leftrightarrow A$. This is part of a class of solutions: a loop. With $v = 2$, we can construct a loop of arbitrary length. The last solution is a long chain of tiles that never loops back around; see Figure 34. This is our first *unbounded* solution — it's a solution where we can always add more tiles. The problem with this solution is that, although it may have $O(n)$ capacity, the *distance* between tiles is *also* $O(n)$. This is not practical since $O(n)$ SPV proofs would be needed to prove state on the far side of the tiling. (The class of solutions that are loops is impractical for the same reason.)

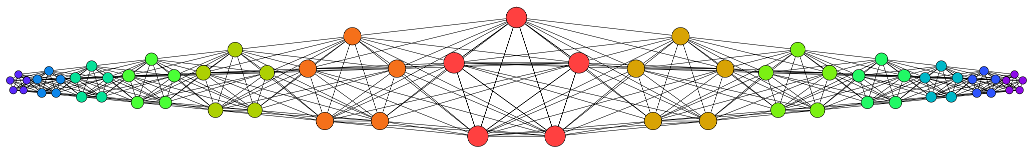


Figure 34: The only unbounded tiling at $v = 2$. Although it is unbounded (we can always keep adding tiles), the *distance* between those tiles is proportional to the size of the network — so this isn't scalable.

Things start to get interesting when $v > 2$.

If a tiling has a valence of $v \geq 3$, then there are many possible patterns of solutions. We will focus on one particular class: the patterns of tiling that are also *trees*.

6.3 Tree-Tilings

The tree-tilings that we'll consider are straight forward to construct. First, we start with a *root tile*. Then, when we need more capacity, we create new tiles that are children of the root tile. Then, when each of these child-tiles is nearing maximum utilization, new children are created as children of *that* tile — they are grandchildren of the root tile. We can continue in this manner indefinitely.

How many children, at most, should we create under each tile? The root tile has *capacity* for v children, and each other tile has capacity for $v - 1$ children and 1 parent. We *could* allow the root tile to have up to v children — and it is possible to create a tree-tiling like this — however, we're not going to do that here. Instead, we'll limit the root tile to $v - 1$ children to match all other tiles.[112] This means that the tilings we'll consider will be n-ary trees — if $v = 3$ then the tiling is a binary tree; if $v = 4$ then it's a ternary tree; if $v = 5$ then it's a quaternary tree, etc. Binary tree-tilings of depths 1, 2, and 3 are shown in Figure 35. Note that, although Figure 35 constructs the tiling as a balanced tree, there's no requirement that a tree-tiling needs to be balanced.

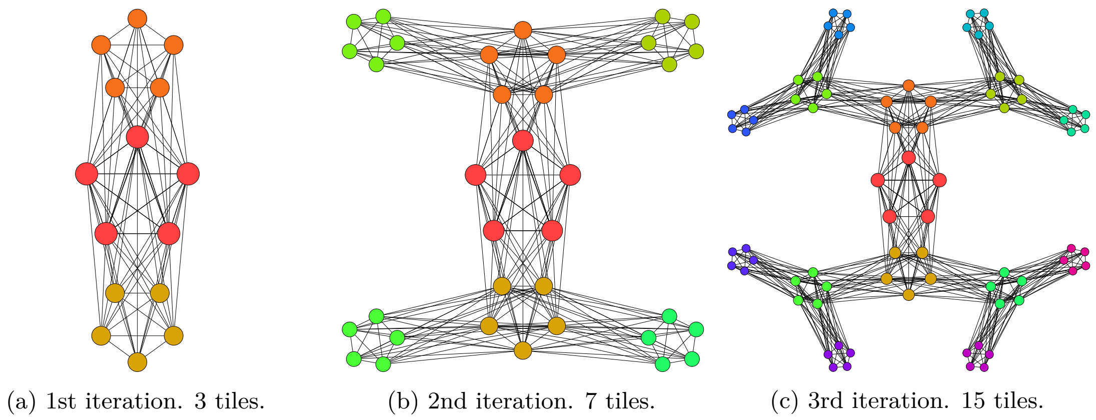

(a) 1st iteration. 3 tiles. (b) 2nd iteration. 7 tiles. (c) 3rd iteration. 15 tiles.

Figure 35: Constructing a binary tree-tiling: the first 3 iterations.

Let's use the term *depth* (d) to describe how far a tile is from the root tile. The root tile has $d = 0$, children of the root tile have $d = 1$, grandchildren of the root tile have $d = 2$, etc.

[112]Exactly why we should do this will be discussed in Section 6.5.

This scaling configuration is $\mathrm{UT}_{\aleph}$, and each of UT_1, UT_2, and UT_3 have a corresponding tiling configuration: $\mathrm{UT}_{\aleph 1}$, $\mathrm{UT}_{\aleph 2}$, and $\mathrm{UT}_{\aleph 3}$.

6.4 Recursive *Proof of Reflection*

So far, we've defined the 'shape' of our tiling, but we don't know much about whether it is (or even can be) secure. Before we cover that in Section 6.5.2, we need to cover a new idea that we are currently missing: *recursive Proofs of Reflection*. The reason for covering recursive PoR first will become obvious, but for now, notice that the leaf tiles in Figure 35c are 3 reflections away from the root tile. Eventually, we will need to answer at least these two questions: *Can the chains in those leaf tiles be secured by the root tile?* and *How do we do this?* Recursive PoR provides the foundation that we need to answer those questions and analyze the security of our constructed tiling.

Let us begin with the simplest case of recursive PoR: 3 chains that do not form a simplex.

6.4.1 Simple Recursive PoR: $L \leftrightarrow M \leftrightarrow R$

Say we have three chains: L, M, and R. They are doing mutual reflection like this: $L \leftrightarrow M \leftrightarrow R$. That is: L reflects M but not R; M reflects both L and R; and R reflects M but not L.

With what we already know, it's clear that M is as secure as all three combined. However, that isn't the case for L and R. How can we make L and R as secure as M *without* having them mutually reflect one another? Is this even reasonable?

Let's consider chain L — what happens if L's local chain-work is, say, $3\times$ that of chain M? The total chain-work that L takes into account is $^4/_3$ of its local chain-work, so an attacker would need $^2/_3$ worth of L's local chain-work to control L's history. This is more than 50% of L's local chain-work, but, depending on R, it's within reason that an attacker might achieve the required hash-rate.

Case 1: If R's local chain-work is large (say, $3\times$ that of M), then M is almost twice as hard to attack as L.[113] This is a problematic situation: it means that an attacker might be able to perform a doublespend against L even though they could not perform one against M. Therefore, this case would not be $O(n)$ secure.[114]

Case 2: However, if R's local chain-work is small (say, $^1/_3$ that of M), then R is not particularly significant. Thus, M is not substantially harder to attack than L, and an attacker could attack both L and M simultaneously (optionally, R too). But, R *is* substantially easier to attack than either M or L. The total chain-work that R takes into account is only $^4/_{13}$ that of the whole network. So, this case is *also* not $O(n)$ secure, even though L *is* $O(n)$ secure.[115]

These two problem cases are what recursive PoR solves.

Notice that in both cases, M is $O(n)$ secure. Given this, chains L and R should be able to use the same work that chain M does to reach the same level of security.

The first problem is that L and R cannot use the same WeightOf algorithm as before (although, M can). Particularly, L and R's WeightOf must count: not only their local chain-work and that contributed by reflected M blocks, but also some additional weight based on reflections from *other* chains. Each M block might record the total chain-work that it adds to M (including PoRs), but L and R's new WeightOf can't use this value directly. Firstly, that value includes past local chain-work that reflected M (which includes both L and R blocks). But more importantly, we don't want to force L and R to verify *every* PoR that contributes to M, just the ones that matter. So how can L and R nodes know which work to count?

[113] M is $^7/_4\times$ harder to attack than L.

[114] If something is "$O(n)$ secure", that means that it is *as difficult to attack as attacking the entire network*. For a particular case to be $O(n)$ secure, each chain in that case must be $O(n)$ secure.

[115] L is secured by $^{12}/_{13}$ of the network. While that is less than 100%, it is not substantially easier to attack than M, so, arguably, the security of L and M is of the same order.

There is a trivial solution to this problem: require L and R nodes to download the headers of *all* relevant chains, verify relevant PoRs, and track the origin of those PoRs.[116] Nodes must track the origin of work contributed via PoR to avoid double-counting work. An easy example of this is: L nodes should not count M's PoRs where an L block is the reflecting block; that work is already directly included in L's local chain-work.

Chain L can only count work contributed by R *after* a multi-stage PoR can be constructed. For example, let's consider the (rather orderly) chain-segments in Figure 36.

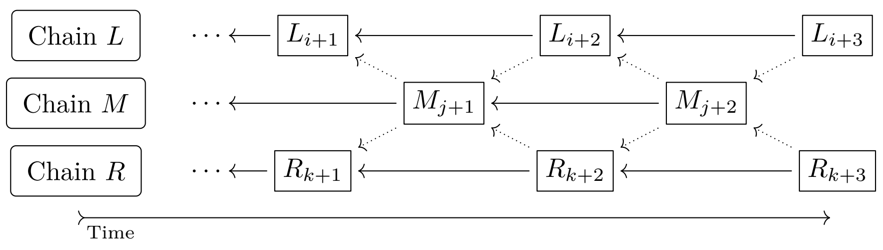


Figure 36: A diagram of chain-segments demonstrating the simplest case of recursive PoR. Solid arrows indicate the usual child-parent relationship, and dotted arrows ($\dashleftarrow$) indicate reflection.

We can see that R_{k+2} *implicitly* reflects L_{i+1} via R_{k+2}'s *explicit* reflection of M_{j+1}. We can use this implicit reflection because the network already depends on all the PoRs that are sufficient to prove it. L_{i+2}'s PoR for L_{i+1} will prove $L_{i+1} \dashleftarrow M_{j+1}$ (i.e., that L_{i+1} was reflected by M_{j+1}). M_{j+2}'s PoR (via R) for M_{j+1} will prove $M_{j+1} \dashleftarrow R_{k+2}$. We combine these two for the full path by which R_{k+2} implicitly reflects L_{i+1}: $L_{i+1} \dashleftarrow M_{j+1} \dashleftarrow R_{k+2}$.

Similarly, we know that L_{i+3} implicitly reflects R_{k+2}: $R_{k+2} \dashleftarrow M_{j+2} \dashleftarrow L_{i+3}$.

Thus, for L_{i+3}, we can construct a full recursive PoR of L_{i+1} via R_{k+2} using the proofs for each link between them: $L_{i+1} \dashleftarrow M_{j+1} \dashleftarrow R_{k+2} \dashleftarrow M_{j+2} \dashleftarrow L_{i+3}$.

Since L nodes already know about all the M and R blocks (and the relevant PoRs), the full recursive PoR *does not need to be explicitly recorded*. It is already known to all L nodes, since each part of the recursive PoR is available in one of the L, M, or R chains (it is explicitly recorded if a +PoRs UT variant is used, and recalculated by the node otherwise).

In Section 2.7, we discussed the idea that a PoR can only contribute to local blocks that are in the reflecting block's past. By inspection, we can see that the most recent R block (the reflecting block) in L_{i+3}'s past is R_{k+2}, and the most recent L block (the local block) in R_{k+2}'s past is L_{i+1}. Thus, we can see that our method for attributing reflected work — following the arrows — works for recursive PoR, too.

There is something important to point out here: *latency*. Notice that these 3 blocks (in chronological order) are required for this recursive PoR to exist: M_{j+1}, R_{k+2}, and M_{j+2}. A non-recursive PoR only needs to wait for 1 reflecting block to be produced (after which the PoR counts as *draft reflected work*). So, in this case, the expected delay will be $3\times$ normal (assuming all chains have the same block periods). In general (if only one PoR path between two given blocks will exist) this delay, measured in block periods, is equal to the length of the full PoR (excluding L blocks), which equals $2d - 1$, with d being the distance (in PoRs) between the chains.

Recapping the first problem: L and R nodes need an expanded WEIGHTOF function with access to *the origin* of implicitly reflected work, and knowledge of *where* that work has already been counted. Nodes can do this relatively efficiently, for all relevant chains, by downloading the headers and verifying the PoRs. This method does not require recursive PoRs to be explicitly recorded in full since the components of these PoRs are already available and have already been individually verified. Our existing rules about how to attribute the weight of PoRs (follow the arrows) still works.

[116]While this solution is trivial, it is also limited. It's fine for a few chains, but in the case of a full tiling, especially with a few layers, we will hit bandwidth limitations at some point. Discussed more in Section 6.4.3.

Our solution to the first problem is promising, but we have used an implicit assumption: there is only one viable path for PoRs between L and R. What if there are *multiple paths* between L and R? This is the second problem.

6.4.2 Recursive PoR with Multiple Paths

Say that we now have four chains: L, M^1, M^2, and R doing mutual PoR like so:

$$\begin{tikzcd}
& M^1 \arrow[dd, leftrightarrow] \arrow[dr, leftrightarrow] & \\
L \arrow[ur, leftrightarrow] \arrow[dr, leftrightarrow] & & R \\
& M^2 \arrow[ur, leftrightarrow] &
\end{tikzcd}$$

Both M^1 and M^2 are already $O(n)$ secure. Assuming that L and R nodes download and verify all PoRs, how can L and R count each other's work?

First, let's consider when L can *definitely* count *all* the work contributed by a R block.

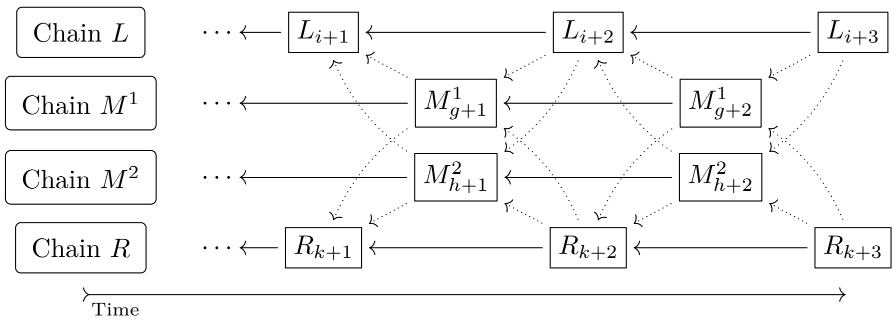


Figure 37: Note that reflections between M^1 and M^2 are not shown.

As before, we'll look at valid PoRs for L_{i+3} that prove L_{i+1} was reflected in R_{k+2}. The exact reflections that exist will be analogous to our prior example, too. See Figure 37 for the chain-diagram. Notice that there are 4 possible paths from L_{i+3} to L_{i+1}:

$$\begin{tikzcd}
& & M^1_{g+1} \arrow[dl, dotted] & & M^1_{g+2} \arrow[dl, dotted] & & \\
L_i & L_{i+1} & & R_{k+2} \arrow[ul, dotted] \arrow[dl, dotted] & & L_{i+3} \arrow[ul, dotted] \arrow[dl, dotted] & L_{i+4} \\
& & M^2_{h+1} \arrow[ul, dotted] & & M^2_{h+2} \arrow[ul, dotted] & &
\end{tikzcd}$$

Here is what we can say about this case: if all 4 of these PoR paths exist, then R_{k+2} *definitely* reflects L_{i+1}, and its weight can be included in L_{i+3}'s total chain-weight.

But, what if only some of those paths are available? Consider this alternative case:

$$\begin{tikzcd}
& M^1_{g+1} \arrow[dl, dotted] & & & M^1_{g+2} \arrow[dl, dotted] & & \\
L_i & L_{i+1} & & R_{k+2} \arrow[ulll, dotted] \arrow[dl, dotted] & & L_{i+3} \arrow[ul, dotted] & L_{i+4} \arrow[dl, dotted] \\
& & M^2_{h+1} \arrow[ul, dotted] & & & M^2_{h+2} \arrow[ull, dotted] &
\end{tikzcd}$$

In this case, multiple L blocks are implicitly reflected in R_{k+2}, and L_{i+3} has two possible recursive PoRs of this: L_{i+1} via M^2_{h+1}; and L_i via M^1_{g+1}. Are both valid? Is either preferable? If L_{i+3} includes the full weight of R_{k+2} in its chain-weight, then what happens when L_{i+4} is mined and the other two PoR paths via M^2_{h+2} become available?

We *can* be sure that R_{k+2}'s weight can be fully included in L's chain-weight *after* L_{i+4} is mined (since all PoR paths now exist). However, it's not so clear what we should do when not all PoR paths yet exist.

Let us approach this from these principles:

- A reflecting block can only contribute work once — we should not count work more than once.
- A PoR contributes as much work as an attacker would need to perform to create a competing PoR of the same weight.

Firstly, we can say that the total work contributed by all possible PoR paths is equal to the sum of local work contributed by each unique block in those paths to their respective chains. That is: no work is counted twice, and all blocks in a PoR path contribute work.

Secondly, consider that the work contributed by an individual PoR is equal to the amount of work that an attacker would need to do to produce an equal and competing PoR. This means that *any* PoR path proving that R_{k+2} reflects L_{i+1} is worth *at least* as much as R_{k+2}. Additionally, a recursive PoR is, in isolation, worth *exactly* as much as the sum of weights of each block in that PoR (excluding local blocks).

Thus, we can safely add R_{k+2}'s *full* weight via L_{i+3} (and that weight is attributed to L_{i+1} since it is the most recent L block in R_{k+2}'s history). An attacker would need to produce at least as much work as R_{k+2} to perform doublespend against L *after* R_{k+2}'s reflection of L_{i+1} has been counted (assuming the doublespend competes with L_{i+1}).

Therefore, only the *first* available recursive PoR via R_{k+2} matters. The number of paths from R_{k+2} to past L blocks is not relevant: L_i is in the history of L_{i+1}, and R_{k+2} can only count once. Additionally, the number of paths from L blocks to R_{k+2} is *also* not relevant, since R_{k+2} is in the history of all such blocks. Having multiple paths does provide some benefit (as there must be some new reflecting blocks, at least initially), but there will almost always be shorter PoRs that provide the same benefit.

Another way to look at multiple paths is that an attacker (who is attacking L, M^1, M^2, and R) does not have to do any additional work to generate multiple redundant paths (besides the additional work that is directly involved in creating new blocks).

Notice that we do *not* require that a recursive PoR use the same chains in each half of the PoR. Of the 4 possible PoRs, the preferable one is: $L_{i+1} \dashleftarrow M^2_{h+1} \dashleftarrow R_{k+2} \dashleftarrow M^1_{g+2} \dashleftarrow L_{i+3}$.[117] We go from L_{i+3} to R_{k+2} via M^1, and from R_{k+2} to L_{i+1} via M^2.

We also can now reason about paths that are longer than the shortest possible path. For example, consider that M^1 and M^2 reflect each other (although this is not shown in our chain-segment examples). There could be PoR paths that include multiple blocks from M^1 and M^2 (e.g., $L_i \dashleftarrow M^1_{g+1} \dashleftarrow M^2_{h+1} \dashleftarrow R_{k+2} \dashleftarrow M^1_{g+2} \dashleftarrow L_{i+3}$). Whilst these kind of paths are valid, they are not that useful: in the example, we expect that M^2_{h+1} already reflects an L block, either L_i directly or a more recent L block.

Finally, let's consider the issue of *latency* again. Now that there are two possible chains to use going from L to R and vice versa, how long should we expect to wait for draft reflected work via R to appear?

After the first L block is generated, we need to wait for a block from either M^1 or M^2. That takes 0.5 block periods on average. Next, we wait for a block from R (1 block period on average), and then for another block from either M^1 or M^2 (0.5 block periods). So the expected waiting time is 2 block periods.

If we had two possible R chains (R^1 and R^2), then we'd expect to wait only 1.5 block periods for draft reflected work from *either* R^1 or R^2. That configuration would look like this:

$$\begin{tikzcd}
 & M^1 \arrow[r, leftrightarrow] \arrow[dd, leftrightarrow] \arrow[ddr, leftrightarrow] & R^1 \arrow[dd, leftrightarrow] \\
L \arrow[ur, leftrightarrow] \arrow[dr, leftrightarrow] & & \\
 & M^2 \arrow[r, leftrightarrow] \arrow[uur, leftrightarrow] & R^2
\end{tikzcd}$$

[117] This recursive PoR path is preferable because it is from the *first possible* L block to enable a PoR via R_{k+2}, and goes to the *most recent* L block in R_{k+2}'s history.

In general, for a recursive PoR of length $2d-1$, where each reflection could be provided by $N_1(v+1)^{-1}$ possible chains, the expected latency of draft reflected work is ${N_1}^{-1}(v+1)(2d-1)$ block periods. We divide N_1 by $v+1$ because each simplex-tile's capacity for reflections is divided equally between v other tiles and itself. This is directly relevant to how fast work propagates within a tiling.

Note that this does not work for a standalone simplex ($d=0 \implies 2d-1=-1$) because we are looking particularly at when recursive PoRs become available (i.e., a draft local block could include them for the first time). In a two-simplex tiling, though (which is equivalent to a standalone simplex), we can see that that $d=1 \implies 2d-1=1$ (each PoR is of length 1), and $v=1$, so the expected latency of draft reflected work from the other tile is $2 \cdot {N_1}^{-1}$ block periods, which, in seconds, is $2 \cdot (N_1 B_f)^{-1} = 2 \cdot \mathbb{C}'^{-1}$ (as expected).

For a leaf tile at $d=3$ where each tile has 5 chains and $v=3$, as in Figure 35c, the expected latency is 1 block period.[118] If, instead, each tile had 50 chains, the expected latency is 0.1 block periods. We will revisit this in Section 6.5.4.

6.4.3 Data Availability

The Axiom of Availability states that miners must download all blocks from all simplex-chains before reflecting the corresponding header. This works fine for small examples like those we've just considered, but, in a large[119] tiling, the sheer volume of data will become a problem at some point.

I don't have a solution for this. In principle, it's not clear that it's even *possible* to solve without resorting to some form of centralization or group effort. The immediate problem with those kinds of solutions is that even if someone else has the blocks, they always have the option of refusing to share them later — verifying that the blocks exist is not enough. These kinds of solutions are substantially different to all the other solutions in this paper and we'll leave these problems for future research.

The good news is that this problem is not tiling specific. Every other network dealing with a lot of data has a similar challenge to solve, and they will hit this wall before UT due to higher validation requirements. It's plausible that if a solution is found, it could work for tiling, too.

Could we use PoS in the outer tiles to mitigate this somehow, similar to dapp-chains? Perhaps, but tiling is a different context to dapp-chains. For one, using a single root token over the entire tiling means that we need stronger guarantees about conservation of coins, cross-chain transactions, etc. Dapp-chains will be created by users for some purpose – we know from the start that there is a party invested in that chain's ongoing operation. Dapp-chains are also somewhat sandboxed and the disappearance of a dapp-chain does not result in a network failure. Simplex-chains in tiles have neither of those properties, and a failure could have disastrous consequences (e.g., if some tiles are severed from the main tiling). So, while PoS could play a role in a solution, it's not obvious what that solution would be.

6.5 Tree-Tiling Security

What does it mean for a tiling to be secure? Well, we have our broad definition from the trilemma: the network is secure against an $O(n)$ attacker. To answer this more specifically, though, we need to first answer: *What does a doublespend against a tiling look like?* Currently, we don't know enough to answer this — we next need answers to questions like *Which PoRs do simplex-chains in a given tile use?* and *How do we govern the difficulties of simplex-chains in different tiles?*

To answer these questions, and then evaluate the security of tree-tiling, we first need to define some key concepts. After that we will construct a security architecture so that we can answer the above questions. Finally, we will analyze this architecture to find out if it's secure.

[118] Note: $N_1 = 20$ in this example, and $N_1 = 200$ in the next.
[119] Particularly, a tiling that has significantly more capacity than a standalone simplex with equivalent parameters.

6.5.1 Key Concepts

6.5.1.1 Local Chain-work, Tile-work, and Unit Tile

We're familiar with *local chain-work* (that's work directly contributed by a single simplex-chain).

Tile-work is the work that is contributed from simplex-chains that are part of a given tile. *Tile-work* is the sum of the local chain-work of each constituent simplex-chain.

We could talk about tile-work directly in terms of *number of hashes*, but this isn't necessary. Instead, we'll normalize tile-work via a *unit tile*. If some tile's tile-work is 3× that of the *unit tile*, then we write down that tile-work as 3 instead of some number of hashes. This will simplify our analysis.

6.5.1.2 The Core

The core of a tree-tiling is *the root tile and its children*. It's called *the core* because all nodes in the network verify the work and PoRs of all simplex-chains that are part of the core. Additionally, the core is where the vast majority of chain-work is done.

6.5.2 Security Architecture

The literal keystone of our tiling is the root tile, which is at depth $d = 0$. Most of the network's total chain-work will be produced by chains in the root tile. We'll guarantee this for $d \geq 1$ (i.e., all other tiles) via the RAA by enforcing an *upper limit* on each tile's tile-work.[120] This upper limit will exponentially decrease as the depth of the tiling increases. Even though tiles at $d \geq 2$ will produce substantially less tile-work than the root tile, they will use recursive PoR to be *at least* as secure as the core.

The purpose of this design is to enable the following argument for $O(n)$ security:

1. Attacking a single chain in a single simplex-tile is at least as difficult as attacking the entire tile and its neighbors (Section 4.10).
2. Attacking any tile is at least as difficult as attacking the core (via recursive PoR).
3. 99% of work is done in the core.
4. Therefore: the core is secure against an attacker with $q < 0.495$.
5. If $q < 0.5$ is $O(n)$ secure, then $q < 0.495$ is also $O(n)$ secure.
6. Therefore: attacking any single chain in any tile is as difficult as attacking the entire tiling, and thus the network is $O(n)$ secure.

6.5.2.1 Inter-tile Security Relationship

Since each tile has $v - 1$ children, we will let $u = v - 1$ for convenience.

We will limit each tile's maximum tile-work production, measured against the unit tile, using the following rules:

1. The root tile produces at least $\mathcal{M}$ units of tile-work ($\mathcal{M} \geq 1$; $\mathcal{M}$ stands for "multiplier").
2. Each non-root tile is limited to r^{-d} units of tile-work ($r > u \geq 2$) where r is a network-wide constant.

The RAA sets the block reward so that each tile produces an appropriate amount of tile-work. The root tile has no upper limit on tile-work production. If a tile would have aggregate rewards greater than its upper limit, then the block rewards of constituent chains are proportionally capped and any excess is allocated to the parent tile. This process continues until all tiles are under their respective limits.

[120] This also means that we're only considering the PoR context of a single root token.

6.5.2.2 Total Work Across a Tiling

How much tile-work is produced over the complete tiling? Let's construct a sum of tile-work over the complete tiling.

The root tile produces $\mathcal{M}$ units of tile-work. At depth $d \geq 1$, each tile produces r^{-d} units of tile-work, and there are u^d such tiles. So at depth d there is at most $(ur^{-1})^d$ tile-work produced. We'll assume that tiles produce the maximum amount of tile-work possible. The sum (S) of all work over the tiling is thus:

$$S = \mathcal{M} + \frac{u}{r} + \frac{u^2}{r^2} + \frac{u^3}{r^3} + \cdots \qquad \text{Sum } \textit{all} \text{ chain-work}$$

$$S - (\mathcal{M} - 1) = 1 + \frac{u}{r} + \frac{u^2}{r^2} + \frac{u^3}{r^3} + \cdots$$

This is a geometric series, and since $\frac{u}{r} < 1$:

$$S - (\mathcal{M} - 1) = \frac{1}{1 - \frac{u}{r}} \qquad \text{Sum of geometric series}$$

$$S - (\mathcal{M} - 1) = \frac{r}{r - u} \qquad \times \frac{r}{r}$$

$$S = \mathcal{M} - 1 + \frac{r}{r - u} \tag{57}$$

6.5.2.3 Doublespend Requirements

A traditional PoW blockchain is secure against an attacker with less than 50% of the network hash rate. In terms of p and q, this threshold is $q < 0.5$.

While we maintained this threshold for simplexes, we will relax is very slightly for tiling. Instead of $q < 0.5$, we will pick a threshold that is very close, but slightly lower. This leaves some wiggle room for us to use PoW in other tiles (outside the core) without weakening the network in a significant way. Choosing $q < 0.495$ (half of 99%) is somewhat arbitrary – in that we don't have a reason to choose it over 0.496 or 0.494 – but it meets our other goals and gives us a nice round number (99%) for the proportion of work done in the core.

For the purpose of this paper, it is assumed that: $\text{UT}_\aleph$ being secure against an attacker with $q < 0.495$ counts as $O(n)$ secure. (That is: $q < 0.495$ is *good enough.*)

6.5.3 Analysis

How can we analyze the security of tree-tilings?

We will consider multiple cases, and in each case establish a *bound* on r in terms of u, q, and $\mathcal{M}$. The intersection of these bounds on r from each of these cases will yield the values of r for which *tiling works* (i.e., it is secure). The value of q that we choose will determine how difficult the root tile is to attack, relative to the entire network — we will choose $q \leq 0.495$ as an acceptable bound on q, so $q = 0.495$ is our threshold. The cases that we analyze will cover all possible positions of a tile in the tiling.

The primary case for us to analyze is that the root tile is secure against an attacker with q proportion of the network-wide hash-rate. Additionally, we need to figure out when a tile (or branch) could be severed from the tiling.

6.5.3.1 The Core is (Almost) as Secure as the Network

Let's constrain the core to being secure against an attacker with up to q proportion of the network-wide hash-rate ($0 \leq q < 0.5$).

The amount of work that attacker is capable of is at most qS (via Equation 57). And the amount of work securing the core is the sum of work contributed by the root tile and its immediate children:

$\mathcal{M} + u \cdot r^{-1}$. The core is not secure when the attacker is responsible for more than half of the work securing the core. Thus, the core is secure when:

$$\frac{1}{2}(\mathcal{M} + \frac{u}{r}) > qS \qquad \text{Assume that the core is secure}$$

$$\frac{1}{2}(\mathcal{M}r + u) > qr(\mathcal{M} - 1 + \frac{r}{r-u}) \qquad \times\, r \text{ and substitute } S \text{ (Equation 57)}$$

$$(\mathcal{M}r + u)(r-u) > 2qr((\mathcal{M}-1)(r-u) + r) \qquad \times\, 2(r-u) \qquad (58)$$

$$r > \frac{u}{2\mathcal{M}}\left(\mathcal{M} - 1 + \sqrt{(\mathcal{M}-1)^2 + \frac{4\mathcal{M}}{1-2q}}\right) \qquad \text{Solve quadratic \& simplify}^{121} \qquad (59)$$

This is our first bound on r. If $u = 2$, $\mathcal{M} = 8$, and $q = 0.495$, then $r > 8$. Therefore, the hash-rate dedicated to the root tile is ~64× that of its immediate children.

6.5.3.1.1 Limiting the Number of Children of the Root Tile

Earlier, we limited the number of children of the root tile (in a tree-tiling) to $v - 1$. This means we can fit twice as many chains in the root tile, and so the hash-rate of the root tile is spread over twice as many chains, and the confirmation rate is doubled (excluding confirmations from external reflections). It also means that hash-rate is less spread out over the tiling, and that the method used above and in Section 6.5.2.2 is relatively simple. Finally, it means that we can grow a simplex to $^1/_2$ of its maximal capacity without committing to any particular details of the tiling scheme (besides the proportion of reflection capacity left for child tiles).

6.5.3.2 Severing Tiles

Is it possible to sever a tile from the tiling? Tiles outside the core have a very small proportion of the network's hash-rate. If all of that tile's miners deliberately censored PoRs with the parent tile, the tile would be severed from the rest of the tiling. If one was severed from the rest of the tiling, performing a double-spend might be trivial.

Assuming $r > 2$, $r > u$, and $d \geq 2$, a tile's chain-work is less than half that of its parent. As we've seen, the amount of work available to the tile via recursive PoR is much greater. Thus, the PoR graph of that tile is easily dominated by any block that can establish recursive PoRs with the core.

Since we're outside the core, the local tile will produce approximately r^{-d} work (in terms of the unit tile), and its children will contribute $ur^{-(d+1)}$ work at most. Note that $ur^{-(d+1)} < r^{-d}$. By comparison, the work contributed by the parent tile is $\mathcal{M} + r^{-1} + r^{-2} + \cdots + r^{-(d-1)}$. Given $\mathcal{M} > 1$, the volume of work contributed via recursive PoRs is at least $^{\mathcal{M}r^d}/_2 > \mathcal{M}r^{d-1}$ times greater than the work contributed by the tile and its children. Therefore, one honest miner in that tiling can prevent severance by producing, at most, 1 block for every $\mathcal{M}r^{d-1}$ blocks produced by the attacker.

If the attacker wishes to avoid this incredible disadvantage, they must attack the parent tile too. This continues recursively until the attacker reaches the core, at which point they must attack the entire network.

If an attacker is attempting to sever a child tile instead, the situation is similar, except that their disadvantage is at least a factor of $\mathcal{M}r^{d-2}$ compared to an honest miner.

Given that honest miners have an overwhelming advantage in these situations, we can skip examining both how this interacts with the DAA and RAA, and the opportunity cost to the attacker of attempting such an attack. Instead, we'll assume that such an attack is impractical, and leave the analysis for future research.

Regarding doublespends, we can conclude that, while it might be possible to disrupt a tile for a short while, all we have to do is wait for recursive PoRs via the core to thwart the attack.

[121] We can quickly verify that Equation 58 and Equation 59 are identical for $u, r > 1$ by graphing them.

6.5.3.3 Analysis Conclusion

We have established bounds where the tiling is secure against severance attacks and doublespends: Equation 59, $u \geq 2$, $r > u$, and $\mathcal{M} > 1$. Practically, values of $u = 2$, $\mathcal{M} = 8$, $r \geq 8$ meet our goals.

6.5.4 Security Propagation Speed / Finalization

Typically, "finalization" refers to a breakpoint whereafter it is practically impossible to revert a transaction.

As we saw in Section 6.4.2, PoR paths form quickly between tiles. Particularly, this depends on the number of chains in each tile, $N_1(v+1)^{-1}$, and the block period.

How long does it take for a block in some tile to be secured by the root tile?

When a block in a tile is produced, miners in the parent tile will reflect it after some propagation delay, ϕ. The expected waiting time is the propagation delay plus the expected time between blocks (of any chain) in the parent tile. After any block in the parent tile has reflected the first block, this process repeats for the parent tile and its parent (i.e., the grandparent of the original tile). Eventually, we reach the root tile, and the reflections begin to propagate back down towards the original tile. (Of course, the reflections propagate in all directions, but we are only concerned with the first path to form between the original tile and the root tile.) We can thus express the expected waiting time until the first recursive PoR with the root tile forms as:

$$\begin{aligned} &\left(\frac{v+1}{N_1 B_f} + \phi\right)(2d-2) + \left(\frac{v+1}{2N_1 B_f} + \phi\right) \\ &= \frac{(v+1)(4d-3)}{2N_1 B_f} + \phi(2d-1) \end{aligned} \tag{60}$$

Ignoring ϕ makes it easy to see how fast this is: given $v = 3$, $d = 10$, $N_1 = 200$ chains, and $B_f = 1/15$ Hz, we can expect a recursive PoR to be available approximately 5.5 seconds after a block is produced. $d = 10$ implies that the tiling has at least 10 layers. Such a tiling has ≥ 1024 *tiles*.[122]

Alternatively, if we set $\phi = 0.5$, then we'd expect the same recursive PoR to be available in $\sim$15 seconds (which happens to be 1 block period in this case).

6.6 Scaling Complexity

6.6.1 Capacity

The tiling is an $(v-1)$-ary tree; assuming it is balanced and has a maximum depth of d, the number of tiles, N_{tiles}, in the tiling is:

$$N_{\text{tiles}} = \frac{(v-1)^{d+1} - 1}{v-2} \tag{61}$$

Each tile in the tiling has $N_1(v+1)^{-1}$ chains – except, of course, the root tile which has twice as many. Thus, the total number of chains in the tiling is

$$\begin{aligned} N_{\text{chains}} &= (N_{\text{tiles}} + 1)\frac{N_1}{v+1} \\ &= \frac{N_1(v-1)^{d+1} - 1}{(v-2)(v+1)} + \frac{N_1}{v+1} \end{aligned}$$

When $v = 3$, we have:

$$N_{\text{chains}} = 2^{d-1} N_1 \tag{62}$$

Clearly, the number of chains depends on how many layers we create, which is determined by demand and systemic constraints.

[122] The tiling needs at least 1 node at $d = 10$, so we assume $d = 9$ is full. In a full binary tree down to a depth of 9, there are $2^{10} - 1$ nodes. $2^{10} - 1 + 1 = 1024$.

Assuming the demand for capacity is $O(n)$, and each chain provides $O(c^i)$ capacity (where $i = 1$ with no dapp-chains, $i = 2$ with 1 level of nesting, etc), we can argue that $O(2^{d-1}) = O(nc^{-i-1})$.

There is at least one physical constraint that will eventually limit us. Assuming all simplex-chains in the tiling use PoW, there will eventually come a layer where we run out of bits to use in the hash. If we had, say, 96 bits to work with (assuming a 256 bit hash), then we'd have $\sim(96 - \log_2(\mathcal{M}))(\log_2(r))^{-1}$ layers to work with.

Given $N_1 = 200$, $\mathcal{M} = 8$, $r = 8$, and $v = 3$, we could support up to $\sim$31 layers, which is more than 10^{11} simplex-chains in total. Granted, there are other constraints that we will be limited by.

6.6.2 Recursive PoRs

Consider some block in some tile outside the core. Given $v = 3$, there will be $N_1/2$ blocks produced in the root tile between this block and the next block. We only need to include one recursive PoR with the root tile to gain the weight of all those blocks — particularly, the recursive PoR goes via the most recent block we know of in the root tile. Also, the other chains in the tile will include recursive PoRs via many of those root tile blocks in the intervening period. In principle, then, we shouldn't need much additional data than these PoRs.

Each merkle (or verkle) branch in such a recursive PoR is $O(\log c)$. There are $O(d) = O(\log(nc^{-i-1}))$ many such branches. Thus the resulting complexity is

$$\begin{aligned} O(\log(c)\log(nc^{-i-1})) &= O(\log(c)(\log(n) - (i+1)\log(c))) \\ &= O(\log(c)\log(n)) \end{aligned}$$

6.6.3 Bandwidth

The primary bandwidth constraint on a full node for some chain in some tile is the PoR graphs for each tile between the local tile and the root (inclusive). Since we've already established that following the PoR graph of a simplex is an $O(c)$ bandwidth load, the overall bandwidth load is

$$O(c\log n - c(i+1)\log c) = O(c\log n)$$

A miner, though, needs to verify the availability of all those blocks – an $O(c^2)$ bandwidth load per tile. Therefore they have an associated bandwidth load of $O(c^2 \log n)$. Assuming that we are using network and chain parameters such that the overall bandwidth requirements of a maximal simplex are tolerable (see Section 5.8), it's not out of the question that multiplying by a factor of $O(\log n)$ is also tolerable. That said, this will be a constraint at some point if the network grows large enough.

If *every* block from *every* tile is downloaded, then we naturally have an $O(n)$ load. However, in smaller tilings we actually *save* bandwidth compared to a maximal simplex of the same capacity. This is due to the simplex tiles themselves being smaller and reducing PoR graph overhead. So, even though the complexities of a tiling are naturally larger than a maximal simplex, there are configurations that increase overall capacity within the same constraints.

7 Attacks

Ultra Terminum — with appropriate configuration — is resistant to the following attacks; see the linked section for discussion:

- 51% Attack, see Section 2, Section 4.8.3, and Section 4.10.
- Selfish mining, see Section 4.7.
- Reflection without publication, see Section 4.1.
- Empty block DoS and censorship, see Section 4.8.
- *Nothing at Stake* and *long range* attacks, see Section 2.6 and Section 3.4.
- Intra-simplex cross-chain attacks, see Section 4.11.

7.1 Dialog: Attacks and Mitigation

Aside This is a fictional dialog between a malicious actor (EMalō) and myself. The factors in play — like the number and types of simplex-chains, the PoW algorithms used, the ROO distribution, etc — were chosen to represent a young UT network. The *real* Amaroo network, when it goes live, will be different. The point of this dialog is to give you an *intuition* for the effect of these factors; it's intended to help answer the question *Why is attacking UT harder and more complex than attacking traditional consensus methods?*

This is the beginning of your direct message history with @EMalō

EMalō I have an offer for you. I'm planning on doing a doublespend. It'll destroy confidence in your system. If you pay me, then I won't do it.

Uhh, okay... IDK if you're credible tho. Convince me and I'll consider your "offer". **Max**

EMalō I have a bunch of sha256 ASICs. I know that one of the simplex-chains uses that, and it doesn't have that much hashing power behind it, so I'm going to attack it. Std stuff.

You're going to mine in private and then publish after enough confirmations? **Max**

EMalō Yeah.

The sha256 simplex-chain only has 6% of the ROO supply on it. Plus, there'll be way too many reflections by the time you try to publish. **Max**

EMalō Wait, what do you mean "reflections", and why does 6% of the supply matter?

Once a block for one chain is published, its existence gets confirmed by the other chains. That's a reflection. There are 15 other simplex-chains, so every sha256 block gets roughly 15 *other* confirmations before the next sha256 block gets produced. The number of confs is a bit random, so mb it's a few less or a few more. The point is that the honest chain will weigh like 15x more than your privately-mined chain, even if your hash-rate is 20x the honest hash-rate. **Max**

The 6% of supply matters b/c the sha256 chain can't have more than 6% of the total security of the system. Whatever work is done on the sha256 chain is always 6% of the total network security. You can mb push out some of the other sha256 miners, but that doesn't help you do a doublespend. It just makes them less profitable for a bit, and you waste money running your miners. **Max**

EMalō

I can still selfish mine.

Max

No you can't; selfish mining works b/c you keep some blocks unpublished. If they're not published then they don't get reflections. The honest blocks have an advantage b/c they're published immediately.

EMalō

I have lots of other ASICs too. And GPUs. I'll mine the ethash chain and the scrypt chain and the cuckoo chain too and just reflect my blocks. Then I'll publish them all at once.

EMalō

> and you waste money running your miners

That's not a problem. The attack will cost you more than it costs me.

Max

Even if you mine those other chains, that's still only 4 out of 16 chains. And they only hold about 30% of the ROO supply anyway. The honest chains will still have like 3-4x the work of your chains, if not more.

Max

Say that you did try to attack *all* the PoW simplex-chains, what about the PoS ones? You need to attack those first if you want to 51% the PoW chains.

Max

> The attack will cost you more than it costs me.

We'll see.

EMalō

I have lots of ROO for PoS chains.

Max

> I have lots of ROO for PoS chains.

GL with that. Since the PoS chains are reflected in the PoW chains, you can't screw with their history to brute-force a favorable quorum — you'd have to rewrite way too much PoW history for that. If you're already PoS mining, then your stake will get slashed the second you publish, so the rest of the network will calc those PoS blocks to have a negative weight.

Max

Not to mention that you can't attack the PoA simplex-chains, anyway. The honest network's always going to have an extra 20% on you.

EMalō

> Since the PoS chains are reflected in the PoW chains, you can't screw with their history to brute-force a favorable quorum — you'd have to rewrite way too much PoW history for that.

Like I said, it'll cost you more than it costs me. Have you forgotten that I can launch these attacks simultaneously?

Max

> simultaneously

That's just it. You can't. Attacking the PoS chains requires set-up, which means that you need to attack the PoW chains *at an earlier point in time* than the moment of launch. But attacking the PoW chains won't succeed without the reflections from PoS chains. And if you want to provide lots of PoS reflections, well you need to attack the PoW chains *even further* back to do *that* set-up. Do you see the problem here? Your only option is to *start from the genesis blocks*.

EMalō I can still attack some dapp-chains.

How? Dapp-chain history is secured by the PoW done in the simplex. If you try, then your stake will get slashed and things will go back to normal. It's more like a donation than an attack. **Max**

EMalō Fine. I'll just DoS your chains instead. Like Luke-Jr did to Coiledcoin.

Did you forget that UT uses a block-dag? The honest nodes will just build on your blocks and include any txs that you don't. It might take a few minutes, but soon enough the honest blocks will weigh too much for you to catch up (b/c they build on your blocks too). That won't help you doublespend, and the DoS vector doesn't work. You're just *adding* to the security of UT, not reducing it. **Max**

EMalō Then I'll make bad blocks. They'll get reflected and will screw with the simplex's history.

No they won't. If you try to get headers reflected without blocks, then other miners will reject them b/c the blocks aren't available — there's no point reflecting them b/c there's no benefit to the miners. If the blocks are invalid, then they will get reflected, but miners on that particular chain will link to them as an invalid uncle. **Max**

So if you make invalid blocks, sure nodes will need to store those blocks for a bit (so that they know they were invalid), but after a while all that is prunable. It's a short term inconvenience that still helps the security of the network long term. Plus you won't get block rewards doing that — so it's actually worse than just making empty blocks. For you, that is. **Max**

Still there? **Max**

✓ BOT Clyde:

Your message could not be delivered. This is usually because you don't share a server with the recipient or the recipient is only accepting direct messages from friends.

8 Conclusions

8.1 Addressing the *Blockchain Trilemma*

Our journey is coming to a close, and it's time we finally answer the question *Does UT solve the Trilemma?* First, let us consider UT_1.

The *decentralization* criterion states that the system must function when each participant has only $O(c)$ resources. We codify this with the chain-parameter k, a generalization of block size, which we define to be $O(c)$. Typically, we've set $k = 3000$ bytes/second since this is close to the real-world value of k for Bitcoin and pre-merge Ethereum, and is therefore, self-evidently, an acceptable value. We've also set $k = 20000$ in some tables so that we have an idea of what we could do if we pushed UT further. The Axiom of Availability means that miners must download *all* blocks — an $O(c^2)$ load. However, miners do not need to *validate* all blocks, or store them for very long; and these blocks aren't required to synchronize with the PoR graph. Therefore, if we fail the decentralization criterion it will be because of bandwidth: ΔS. But, what is ΔS? Way back in Section 1.1.2, I suggested a test for whether UT_1 passes or fails in this sort of situation: is this a bottleneck or is there excess capacity? In Section 5.10 we saw that, when $k = 3000$, reasonable network parameters meant $\Delta S \in [293, 2188]$ kB/s. This equates to $\Delta S \in [759, 5184]$ GB/month. Currently, Digital Ocean offers[123] a VPS with 5 TiB/month (outbound) for 48.00 USD. By inspection, this is $O(c)$, so, at $k = 3000$, the ΔS requirements on miners is not a bottleneck. Therefore, with appropriate parameters, UT_1 passes the decentralization criterion.

The *security* criterion states that the system must be secure against an attacker with $O(n)$ resources. The standard way to check if a PoW system passes this criterion is whether it is resistant to a 51% attack. We saw in Section 4.10 that UT_1 is indeed secure against a 51% attack. We also saw in Section 4.11 that cross-chain transactions within the simplex were as secure as the network. Therefore UT_1 passes the security criterion.

The *scalability* criterion requires that the system can process more than $O(c)$ transactions. Given $k = 3000$, $\Sigma\mathrm{TPS}_1 \in [10^3, 8.4 \times 10^3]$. By inspection, 1000 TPS is more than $O(c)$. Therefore UT_1 passes the scalability criterion.

Thus UT_1 satisfies all 3 trilemma criteria *as put by Vitalik.*

Aside

At $k = 20000$ B/s, $\Delta S \in [10, 93]$ MB/s, or $\Delta S \in [25, 241]$ TB/month. This is definitely beyond many ordinary users, but would work for miners at some economy of scale. (241 TB costs ~2235 USD/month[a] from Digital Ocean.) Such a simplex would have $N_1 \in [10^3, 10^4]$ chains, and $\Sigma\mathrm{TPS}_1 \in [4 \times 10^4, 3.8 \times 10^5]$ TPS.

[a]Jan 2025; see https://docs.digitalocean.com/platform/billing/bandwidth/

Second, let's consider UT_2 and UT_3.

On the assumption that a secure $O(c)$ PoS chain exists, then using that for dapp-chains provides an $O(c)$ bump to scalability. The additional computational burden to a miner or full node is negligible, so UT_2 satisfies the decentralization criterion. The security of the simplex isn't affected, and the dapp-chain is more secure than the PoS chain would be in isolation, so UT_2 satisfies the security criterion. Since the dapp-chains are PoS, their effective header size will be greater than what is used for the calculations in Section 5.10. Assuming that the effective header size of dapp-chains is $10\times$ larger than in Section 5.10 (and $k = 3000$, of course), we see that $N_2 \in [5 \times 10^3, 4 \times 10^4]$ dapp-chains. We also see that $\Sigma\mathrm{TPS}_2 \in [6 \times 10^4, 5 \times 10^5]$. By inspection, 60,000 TPS is a lot. Therefore, UT_2 satisfies all 3 trilemma criteria *as put by Vitalik.*

On the assumption that a secure $O(c^2)$ PoS chain exists...you know where this is going. If UT_3 works, then we're talking millions of TPS (~10^6 to 10^7) at $k = 3000$ B/s.

Share security, and *capacity follows.*

[123]See https://www.digitalocean.com/pricing/droplets (also via web.archive.org however client-side js breaks the page shortly after loading). Note: inbound data is free.

8.2 Addressing the Stronger Trilemma

In Section 1.1, I took issue with the scalability criterion and suggested a stronger version: $O(n)$ transactions in $O(1)$ time.

How can we reasonably estimate what an $O(n)$ transaction load looks like? There are approximately 8 billion people on Earth. If everyone one of them made 1000 transactions per day, the network would require 8×10^{12} transactions per day, which is about 93 million ($\sim 10^8$) TPS. Clearly, this is greater than any of the TPS values we calculated above.

For $\text{UT}_{1+\text{HOT}}$ to reach 10^8 TPS, we'd need a large $k > 20000$ and maybe a longer block period.

$\text{UT}_{2+\text{HOT}}$, by comparison, is already around the $\Sigma\text{TPS}_2 \approx 5 \times 10^5$ mark with our standard chain parameters. If $B_f = 1/60$, then $\Sigma\text{TPS}_2 \approx 10^7$; however, ΔS would be around $5\times$ larger. That is a substantial increase in bandwidth requirements, and pushing to 10^8 TPS is not really practical within $O(c)$ constraints.

Alternatively, we can look to tiling for extra capacity in the first level of nesting. We found in Section 6.5 that tree-tilings can be constructed to be $O(n)$ secure, so all that remains to show is that $\text{UT}_\aleph$ is $O(n)$ scalable *and* decentralized *at the same time*. The big problem for arbitrarily large tilings is block availability. Smaller tilings with acceptable ΔS are still useful, though.

$\text{UT}_{\aleph 1+\text{HOT}}$ reaches 10^8 TPS around 15 layers ($d \leq 15$) and would have about 10^7 chains. If block availability in tilings remains an unsolved problem, then this is clearly not practical.

On the other hand, $\text{UT}_{\aleph 2+\text{HOT}}$, with standard chain parameters, reaches $\Sigma\text{TPS}_2 \approx 10^8$ at around 8 layers; a $\sim 100\times$ increase in the number of total chains (and ΔS). This is much closer to reasonable parameters.

Is it so close that, perhaps, an otherwise unacceptable technique could be used to ensure the availability of all simplex-tile blocks? An example of such a technique might be some kind of division of responsibility between miners so that they only need to ensure blocks from tiles above and below the given tile are available (in addition to blocks from that tile, of course). For all tiles other than the root tile, this would dramatically reduce ΔS to about $2^{-d}\times$ what it would otherwise be, where d is the depth of the tile.

So, *if* the problem of block availability in a tiling can be and is solved, we are clearly $O(n)$ scalable and decentralized with any tiling. However, since it is unsolved, $\text{UT}_{\aleph 2}$ is *not currently* $O(n)$ scalable *and* decentralized, but could potentially reach that in the future.

Finally, what about UT_3 and $\text{UT}_{\aleph 3}$?

A conservative estimate of UT_3's TPS with standard chain parameters is around 10^7. Given the sensitivity of this number to small changes in chain parameters, we could easily tweak things a little to reach 10^8 TPS.

$\text{UT}_{\aleph 3}$, instead of tweaking parameters, could increase capacity over UT_3 by $4\times$ via tiling to just brush these kinds of numbers with only 3 layers (the tiling shown in Figure 35c). In turn, ΔS also increases by $4\times$ what it would be for a maximal simplex with the same chain parameters.

Arguably, if the UT_3 chain parameters are practical, then this is practical. Therefore, with carefully chosen parameters, $\text{UT}_{\aleph 3}$ is secure, $O(n)$ scalable, and decentralized *at the same time*.

8.3 Terminus Est

What a ride, ey? Let's recap.

We started from a simple idea: that a blockchain confirming the history of another blockchain was worth *something*. From there, we constructed *Proof of Reflection* — a new consensus primitive that facilitates the summation of chain-work from different chains. We then found ways to convert work between chains that is compatible with our goals and context, allowing otherwise incompatible consensus methods to cooperate. Figure 38 shows how it is the missing idea that addresses the core conflict of the *Blockchain Trilemma*.

Mutual PoR plus the conversion of work allowed us to construct *the Simplex* and UT_1 — a highly interconnected network of blockchains with $O(c^2)$ capacity. By utilizing one-way PoR in combination with existing blockchain designs, we added dapp-chains to construct UT_2 and bump the overall capacity to $O(c^3)$. Based on the observation that some current PoS networks claim to be $O(c^2)$ scalable, we used these designs as dapp-chains instead to construct UT_3 and reach $O(c^4)$ capacity.

We then explored the practical side, and found that we needed the Axiom of Availability to ensure reflections between simplex-chains only included available simplex-blocks. As we looked into the specifics of how we would structure blocks and PoRs, we found several optimizations that significantly reduced both the computational and bandwidth requirements for network participants. We also discussed how to support stateless clients and fraud proofs – tools we would need later. This ultimately led us to construct the PoR graph and the Axiom of Maximal Reflection. We adapted NIPoPoWs into NIPoPoWRs, which work for the PoR graph instead of a standalone blockchain. We then integrated block-DAGs to prevent minority attackers from performing effective DoS or censorship attacks, and developed NIPoPPoWs which work with block-DAGs. We realized that we now had the conditions to *require* miners to build on the latest block(s) without any appreciable downside, and enshrined this as the Axiom of Unified Ancestry. Throughout this, we found that these new constructions lacked multiple weaknesses of traditional chains, even though solving those problems was not a goal. Selfish Mining and PoS long range attacks are two such examples. Somewhat surprisingly, we also found that the confirmation rate of a simplex is *inversely proportional to c*! Not only is this faster than any existing network, but the dynamics of block production cause *miner resonance*, which acts to synchronize miners and reduces the variance in block production times in PoW simplexes. Using our newfound knowledge, we tested the *Confirmation Equivalence Conjecture* and found it consistent with reality. Taking all of this, we devised a fast, succinct, secure, and reliable method for intra-simplex cross-chain transactions. As icing on the cake, we developed *expedited transactions* that not only allow for near-zero time-to-first-confirmation, but also allows a chain to dynamically expand its capacity by up to $\sim 5\times$ to heavily mitigate the negative effects of congestion. And with that, we completed the design for UT_1 and the Simplex.

We analyzed this design more formally, and found that UT_1 was indeed an $O(c^2)$ construction. We also saw how every relevant aspect of UT_1 was within $O(c)$ constraints, except for ΔS, which was $O(c^2)$. However, when we checked the *actual numbers*, we found ΔS was within reasonable limits and is easily accommodated by today's standards. At this point we were already out-scaling any existing blockchain design, but that didn't stop us confirming the scalability of UT_2 and UT_3, and calculating some girthy capacity estimates as a result.

After all this, we reconsidered how simplex-chains might be organized and discovered *Recursive PoRs* and *Simplex Tilings*, with a particular focus on tree-tilings. We developed a security model for tree-tilings that was secure against an attacker with up to 49.5% of the network hash rate. Were it not for the bandwidth requirements on miners imposed by the Axiom of Availability, this would be the ultimate scaling construction as it is practically unbounded — we could always add more tiles, growing the capacity exponentially with the depth of the tiling. We don't know if that availability problem is solvable, but even if it isn't, we saw how smaller tilings are still useful.

Penultimately, we evaluated UT_1 against the *Trilemma* and found that it satisfied all three criteria as originally put. But that was not enough, so we examined each scaling configuration of UT against a stronger trilemma. Although boasting impressive capacity (see Table 13, Table 14), most configurations fell short of what we set as the standard for $O(n)$ scalability: 100 million TPS. But! We did find one configuration that definitely worked: $\mathrm{UT}_{\aleph 3}$.

With that, I think it's finally time to make *the* claim.

The *Blockchain Trilemma* is solved. ■

[124] If the block availability problem can be solved for tilings, then all tilings are $O(n)$ scalable since their capacity is unbounded. If it cannot, tilings are still useful but do not provide $O(n)$ scaling on their own.

[125] $\mathrm{UT}_{\aleph 3}$ is decentralized if the maximum tiling depth is kept low. As we found earlier, $\mathrm{UT}_{\aleph 3}$ with $d \leq 3$ provided a tolerable ΔS, and lead to a network-wide TPS of around 10^8, even though we assumed dapp-chain headers to be $10\times$ worse than the theoretical maximum (dapp-chain headers would be $\sim$1000 bytes). *This specific case* is what "Yes" refers to.

Table 12: Table evaluating UT against trilemma criteria. TPS ranges are the minimum and maximum values for $3000 \leq k \leq 20000$ B/s across all UT variants. Tilings are not limited in d.

UT Config.	Decentralized?	$O(n)$ Secure?	$O(n)$ Scalable?	
UT_1	Yes	Yes	Unlikely	(Max. TPS: $\sim$ 1K - 400K)
UT_2	Yes	Yes	Possibly	(Max. TPS: $\sim$ 60K - 2B)
UT_3	Yes	Yes	Probably	(Max. TPS: $\sim$ 10M - 6T)
$UT_{\aleph 1}$	Maybe[124]	Yes	Probably	(Max. TPS: $\sim$ 50M - 10^{18})
$UT_{\aleph 2}$	Maybe	Yes	Yes	(Max. TPS: $\sim$ 30B - 10^{21})
$UT_{\aleph 3}$	Yes[125]	Yes	Yes	(Max. TPS: $\sim$ 10^{12} - 10^{25})

Table 13: UT capacity vs. Earth with standard chain parameters and heavy dapp-chains headers. Parameters: $k = 3000$ B/s; $B_f = 1/15$ Hz; $B_h = 112$ B; $D_f = 1/15$ Hz; $D_h = 560$ B. *Per capita* here assumes a population of 10 billion people. In all cases, the UT variant is +HO. Tilings have $d = 3$. ΔS: BW req. for miners; Δs: sync BW req. for full nodes (PoR graph + 1 simplex-chain).

UT Config.	Σ TPS	Tx/day/capita	Δs (kB/s)	(TB/mo)	ΔS (kB/s)	(TB/mo)
UT_1	4,219	0.036	10.8	0.028	1,097	2.884
UT_2	339,007	2.929	10.8	0.028	1,097	2.884
UT_3	27,241,610	235.368	10.8	0.028	1,097	2.884
$UT_{\aleph 1}$ $(d = 3)$	16,875	0.146	43.1	0.113	4,386	11.535
$UT_{\aleph 2}$ $(d = 3)$	1,356,027	11.716	43.1	0.113	4,386	11.535
$UT_{\aleph 3}$ $(d = 3)$	108,966,438	941.470	43.1	0.113	4,386	11.535

Table 14: UT capacity vs. Earth, as in Table 13 with $k = 6000$ B/s (3.5× Bitcoin's); $D_h = 1024$ B.

UT Config.	Σ TPS	Tx/day/capita	Δs (kB/s)	(TB/mo)	ΔS (kB/s)	(TB/mo)
UT_1	16,875	0.146	22.1	0.058	4,373	11.500
UT_2	1,483,154	12.814	22.1	0.058	4,373	11.500
UT_3	130,355,358	1,126.270	22.1	0.058	4,373	11.500
$UT_{\aleph 1}$ $(d = 3)$	67,500	0.583	88.5	0.233	17,492	45.999
$UT_{\aleph 2}$ $(d = 3)$	5,932,617	51.258	88.5	0.233	17,492	45.999
$UT_{\aleph 3}$ $(d = 3)$	521,421,432	4,505.081	88.5	0.233	17,492	45.999

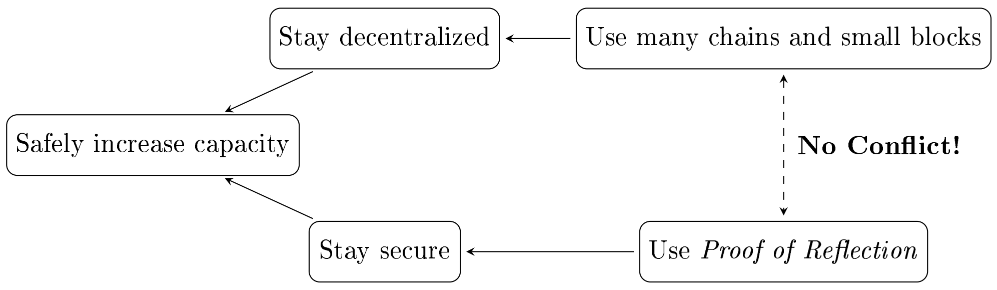


Figure 38: A solution to the core conflict of the *Trilemma*.

9 Criticisms of UT

This section documents the major reasons that UT might not work — or, equivalently, the most valuable contributions. I have also provided some commentary on the criticisms.

There are no known decisive criticisms of UT_1.

9.1 Open Problems

9.1.1 Dapp-chains & PoS

Main content: Section 3.4.

Criticism: Dapp-chains are based on the assumption that a secure PoS method exists when adapted for one-way PoR. No specific PoS method is provided and it's not clear that PoS + PoR actually works without a clear design for such a system. The other claims about dapp-chains and UT_2 scaling and above is contingent on a suitable PoS method being found or created. PoS is insecure when used in isolation and it isn't clear that PoS can be secure, even with the help of PoR and a purely PoW simplex. Even if a suitable method of PoS is found, PoS + PoR is inefficient compared to PoW + PoR and some capacity estimates are therefore overly optimistic.

Commentary: This criticism is correct if PoS is never secure, although it is not the complete picture. There are some cases that work — PoA dapp-chains and non-financial dapps, for example. However, the goal is a universal method that works for any dapp-chain and allows for secure cross-chain transactions between the simplex and dapp-chains. And while we don't have an otherwise-working PoS + PoR implementation, we also have no reason to believe that conversion is impossible or that PoR is incompatible with PoS.

If PoS + PoR is secure in a PoW context, there is no other fundamental barrier to UT_2 or UT_3. Thus this criticism is possibly refuted *if and only if* PoS + PoR to PoW is secure.

9.1.2 $\text{UT}_{\aleph 1}$ Block Availability & $O(c^2 \log n)$ Bandwidth

Main content: Section 6.4.3 and Section 6.6.3.

Criticism: Without a solution to block availability, $\text{UT}_\aleph$ doesn't work for $d \gtrsim 4$. It works to expand capacity by up an order of magnitude but not beyond. Even that might not work because the security model is untested and there are no comparable PoW-based real-world examples to examine. $O(c^2 \log n)$ bandwidth for validating nodes is too high. $O(nc^{-i})$ bandwidth for root-tile miners is too high.

Commentary: While $\text{UT}_\aleph$ is useful, it is not a complete solution to $O(n)$ scaling. That it *is* useful hints that there might be more and better PoR-based geometries to discover. If a satisfactory solution to block availability can be found then it will also solve the bandwidth problem and the bottleneck will be the *gradient of work* required as distance from the core increases ($d_{\text{MAX}} \in [10, 30]$).

9.2 Mitigated Risks

9.2.1 A Key Point for UT_1 Is Flawed

Risk: Several key components are required for UT_1 to work. There are no *known* decisive criticisms but maybe you can find one? PoR and Conversion of Work (Section 2), Simplex Security (Section 4.10.1), and Intra-Simplex Cross-Chain Transactions (Section 4.11) are three such components.

9.2.2 UT_1 $O(c^2)$ Bandwidth

Main content: Section 4.1.

Risk: Whether ΔS is tolerable depends on k (block size) and is mitigated with a suitable k (see Table 13, Table 14). A k of 3000 B/s has plenty of breathing room, and 20,000 B/s is on the high end of what could possibly work. For comparison, Bitcoin has $k \approx 1700$ B/s.

Notation

Table 15: Notation defined in this document.

Term	Definition	Unit
c	Abstract representation of per-node computational resources.	-
n	Abstract representation of network size.	-
p	Proportion of the network's hash-rate controlled by honest nodes.	-
q	Proportion of the network's hash-rate controlled by the attacker.	-
L_d, R_d	The difficulty of chain L/R.	hashes/block
L_r, R_r	The block reward of chain L/R.	coins/block
w	Some amount of *work*.	hashes
$X_{R \to L}$	Exchange rate between L-coins and R-coins.	L-coins/R-coin
$\mathbb{C}'$	Confirmation rate.	Hz
g	Hash digest size.	bytes
k_i	A generalization of block size: the average per-chain raw throughput at the i^{th} level of nesting.	bytes/second
k	Average per-chain raw throughput across nesting levels. k is used to simplify reasoning and equations, esp. when all k_i are equal.	bytes/second
T_i	Network throughput at the i^{th} level of nesting.	bytes/second
N_i	Number of chains at the i^{th} level of nesting.	chains
N_{tiles}	Number of tiles in a simplex-tiling.	tiles
B_f, L_f	Base-chain block frequency.	Hz
B_h	Base-chain header size.	bytes
D_f	Dapp-chain block frequency.	Hz
D_h	Dapp-chain header size.	bytes
ϕ	Propagation delay across the network	seconds
Σ TPS$_i$	Network-wide transactions per second at the i^{th} level of nesting (given no additional levels).	tx/s
Δs	Minimum network bandwidth for a full node to remain in sync with a single simplex-chain (whilst also validating PoRs).	bytes/second
Δr	Minimum network bandwidth for a full node to fully reconstruct the PoR graph.	bytes/second
ΔS	Minimum network bandwidth for a *mining* (or fully validating) node to ensure block availability over all reflected simplex-chains.	bytes/second
DAA$_N$	The number of blocks over which the DAA operates.	blocks

Nomenclature

Table 16: Nomenclature defined in this document.

Term	Definition	Reference
UT_i	The UT scaling configuration with i levels of nesting.	Section 3
UT_1	UT with base-level chains only — $O(c^2)$ scalability.	Section 3.2
UT_2	UT with nested dapp-chains — $O(c^3)$ scalability.	Section 3.4
UT_3	UT with nested dapp-dapp-chains — $O(c^4)$ scalability.	Section 3.4
$\text{UT}_{\aleph i}$	Tiling of UT_i — $O(n)$ scalability.	Section 6
+PoRs	Protocol variant: explicit proofs.	Section 4.2
+PoRTs	Protocol variant: explicit proofs + T.	Section B.1.1
+HOPoRs	Protocol variant: explicit proofs + header omission.	Section 4.4
+HOPoRTs	Protocol variant: explicit proofs + header omission + T.	Section B.1.1
+OP	Protocol variant: omitted proofs.	Section 4.2
+OPT	Protocol variant: omitted proofs + T.	Section 4.4
+HO	Protocol variant: omitted proofs + header omission.	Section 4.4
+HOT	Protocol variant: omitted proofs + header omission + T.	Section 4.4

Glossary

Axiom *Network axioms* are foundational rules expressed as a principle and predicate. Consensus-forming nodes must adhere to them. See definitions: Availability, Maximal Reflection, Unified Ancestry. 49

Base-chain A chain that has no parent-chains; i.e., is at the base nesting level. 105

Confirmation Equivalence Conjecture (CEC) The conjecture that, when using PoR and appropriately converting work, confirmations of reflecting chains can be treated as *equivalent* to local confirmations of the same weight. See Equation 33. 79

Convertible Context The boundary of a group of values that are mutually convertible. Within a convertible context, all values must be of the same *scale* or have known exact scaling factors. 24

Dapp-chain An *application-specific* child-chain that is secured via the parent-chain. Dapp-chains may have architecturally distinct state- and transaction-schemes (distinct from those schemes used in the simplex, and other dapp-chains). 40

Difficulty Adjustment Algorithm (DAA) An algorithm which updates its chain's difficulty as valid blocks are produced. The *output* of a DAA is *context laden* — units take on *additional context*. 23

Explicit Proofs (+PoRs) The UT protocol variant wherein miners/validators explicitly record *both* reflected headers *and* the single missing merkle branch required to prove reflection. 50

Foundational Consensus Mechanisms Those mechanisms, like PoW and PoS, which can work in some *standalone* fashion; PoR is a cross-chain *extension* to such mechanisms. 34

Fraud Proof Cryptographic evidence that a transaction, block, or state transition was incorrect. 54

Freedom of Incentivization The property whereby child-chains have free choice of incentive-system (i.e., the nature and dynamics of their root token, or lack thereof). 44

Freedom of Protocol The property whereby child-chains have free choice of protocol (including consensus mechanism, scripting, accounting methods, block structures, etc). 44

Hash Truncation (+T) The UT protocol variant wherein miners/validators refer to reflected headers using *only* the least significant half of the hash. This effectively halves the hash size in throughput calculations for +OP and +HO variants. 53

Header Omission (+HO) The UT protocol variant wherein miners/validators explicitly record *only* the hashes of reflected headers. A requirement is that block producers must eagerly download the headers of all simplex-chains and deterministically recalculate the relevant Proofs of Reflection. 51

Header-transactions Dapp-chain headers that are encoded as simplex-level transactions; i.e., they are processed by a simplex-chain as a transaction, but they also function as the header for a dapp-chain block. 40

Main Chain In a block-DAG, the *main chain* is the continuous chain of primary parents from the best block to the genesis block. Blocks that are part of the main chain are *on-main*, and blocks that are not are *off-main*. 68

Maximal Simplex A simplex with the maximum TPS under given $O(c)$ constraints. 106

Miner Resonance The effect whereby block production *variance* is reduced when miners can (and do) collectively change which chain they are currently mining faster than blocks are produced for those chains, due to changes in network-wide incentivization. 78

Omitted Proofs (+OP) The UT protocol variant wherein miners/validators explicitly transmit *only* the reflected header component of PoRs, such that necessary proofs of reflection themselves are deterministically recalculable. 50

Projection A *projection* of a chain is its *headers-only* version that has been recorded and evaluated *by a different chain.* For example, BTC Relay is a smart contract by which Ethereum previously hosted a *projection* of Bitcoin. The *act* of one chain creating and maintaining the projection of another is called *imaging*. 9

Proof of Reflection (PoR) The consensus technique whereby a blockchain becomes more difficult to attack by including work done by reflecting blockchains in its *fork rule*. 15

Reward Adjustment Algorithm (RAA) An algorithm which updates the block reward of each chain in a *network* of chains that share a root token. Similar to a DAA, the *output* of an RAA is *context laden*. 28

Root Token (RT) *aka* ***Coin***. The typically sole network-level token required by blockchain protocols. e.g., Bitcoin has BTC, Ethereum has ETH, Polkadot has DOT, Cardano has ADA, Amaroo has ROO, etc. 20

Scaling Factor Also: "Scale $\times$". For a given k, it is the factor by which TPS increases with an additional nesting level. In effect, it allows for comparison of the efficacy of scaling schemes when k is fixed. For some designs, the *Scaling Factor* can change between nesting levels. 143

Simplex The single coherent structure that emerges from a collection of blockchains that mutually reflect each other. 38

Simplex Tiling An interconnected graph of mutually reflecting simplex-tiles. 115

Simplex-chain A blockchain that is part of a *simplex*; it mutually reflects all other simplex-chains in that simplex. 38

Simplex-Tile A simplex which partitions its capacity for mutual reflections such that each simplex-chain mutually reflects all other simplex-chains in that tile, and all other simplex-chains in *neighboring* (or *adjacent*) tiles. 115

List of Figures

List of Tables

Key References

1. Bitcoin: A Peer-to-Peer Electronic Cash System, Satoshi Nakamoto, 2008.
2. Analysis of hashrate-based double-spending, Meni Rosenfeld, 2012.
3. Inclusive Block Chain Protocols, Lewenberg, Sompolinsky, Zohar, 2015.
4. Ethereum Wiki: On sharding blockchains FAQs, Vitalik Buterin, 2017.
5. Non-Interactive Proofs of Proof-of-Work, Kiayias, Miller, and Zindros, 2018.
6. Critical Fallibilism Course, Elliot Temple, 2020.
7. Multi-Factor Decision Making Math, Elliot Temple, 2021.

Supplementary References

1. Secure High-Rate Transaction Processing in Bitcoin, Sompolinsky, Zohar, 2013.
2. An Economic Analysis of Difficulty Adjustment Algorithms in Proof-of-Work Blockchain Systems, Noda, Okumura, Hashimoto, 2020.
3. Segregated Witness, Bitcoin Wiki, 2017 (edit 2021).
4. BIP-141, Lombrozo, Lau, Wuille, 2015.
5. Polkadot Whitepaper, Gavin Wood, 2016.
6. GRANDPA: a Byzantine Finality Gadget, Stewart, Kokoris-Kogia, 2020.
7. Ouroboros: A Provably Secure Proof-of-Stake Blockchain Protocol, Kiayias, Russell, David, Oliynykov, 2019.
8. Ethereum Yellow Paper / Petersburg Version 41c1837, Gavin Wood, 2021.
9. Majority is not Enough: Bitcoin Mining is Vulnerable , Eyal, Sirer, 2013.
10. Ethereum - BTCRelay, Ethereum Contributors, 2017.
11. Luke Jr's attack on Coiledcoin, Luke-Jr, 2012.
12. The Defence of PoS, Zack Hess, 2021.
13. Proof of Stake, Zack Hess, 2021.
14. Stateless Full Node, Zack Hess, 2021.
15. The Stateless Client Concept, Vitalik Buterin, 2017.
16. Making UTXO Set Growth Irrelevant With Low-Latency Delayed TXO Commitments, Peter Todd, 2016.
17. Merkle Mountain Range, Peter Todd, 2018.
18. Merklix tree, deadalnix, 2016.
19. T.E.T.O Draft, Paul Firth, 2016.
20. DagCoin Draft, Demian Lerner, 2015.
21. IOTA BitcoinTalk Thread, Come-From-Beyond, 2015.
22. Instant Transactions with a stream of work – Quanta, Max Kaye, 2015.
23. For Public Consideration: [Marketcoin | MKC] A P2P Trustless Cryptocoin Exchange, Max Kaye, 2013.
24. [RFC] Marketcoin - A web of distributed markets with a common unit, Max Kaye, 2014.
25. Coppr: Chainheaders, Max Kaye, 2014.
26. What are NIPoPoWs?, nipopows.com, 2019.
27. OpenEthereum, Parity, 2022.

A Comparison: "The Big 4"

The following comparisons (Table 17, Table 18, and Table 19) are intended to be an *apples to apples* comparison between UT variants and "The Big 4": Bitcoin, Cardano, Eth2, and Polkadot. Cardano, as a cutting-edge network, is an exception here: the Cardano/IOHK teams have not been pursuing *Layer 1* scalability solutions and are focusing instead on a *Layer 2* solution: Hydra via EUTXOs. Ethereum is also an exception of sorts: these comparisons use the original *Eth2* sharding roadmap as an example of a sharded Layer 1, whereas the Ethereum network itself is now pursuing a rollup-centric scaling strategy via their beacon chain. I mention this particularly because the nature of an *apples to apples* comparison casts Cardano and Ethereum in a light that some might consider to be misleading. However, these comparisons deliberately do not consider *Layer 2* scalability solutions for one very simple reason: all networks can implement them in some fashion. It is not a fair (or accurate) comparison *of blockchain architecture* if *Layer 2* scaling solutions are considered for some networks and not for others.

In addition to those four, a network named *Opt.Shard* (for *Optimal Sharding*) is included in these comparisons. This is a theoretical network which uses the *best parameters possible among UT configurations*. No real-world sharded network has come close to this level of performance, and it is incompatible with PoS.

Term

Scaling Factor: Also: "Scale ×". For a given k, it is the factor by which TPS increases with an additional nesting level. In effect, it allows for comparison of the efficacy of scaling schemes when k is fixed. For some designs, the *Scaling Factor* can change between nesting levels.

Table 17: A comparison of quantitative scaling properties between UT and various networks given $k = 3000$ bytes/s. Transaction size is set to 250 bytes, $D_f = B_f$, and $D_h = B_h$. Note: '∞' here does not literally mean infinite; however, we cannot calculate concrete values without specifying parameters arbitrarily which are otherwise unbounded.

k, B_f, B_h	Network	E. B_h (B)	E. D_h (B)	Scale ×	N_2	Σ TPS
$3000, 1/600, 80$	Bitcoin	80	-	1	-	12
$3000, 1/20, 1070$	Cardano	1070	-	1	-	12
$3000, 1.58, 144$	Solana	144	-	1	-	12
$3000, 1/15, 84$	$\text{UT}_{1+\text{PoRTs}}$	99	-	1	-	1,363
$3000, 1/15, 84$	$\text{UT}_{1+\text{HOPoRTs}}$	56	-	1	-	2,319
$3000, 1/15, 84$	$\text{UT}_{1+\text{HOT}}$	16	-	1	-	8,437
$3000, 1/6, 288$	Polkadot	288	819	21	21	263
$3000, 1/12, 200$	Eth2	200	312	115	115	1,384
$3000, 1/15, 66$	Opt.Shard	66	66	681	681	8,181
$3000, 1/15, 84$	$\text{UT}_{2+\text{PoRTs}}$	99	66	681	77,479	929,750
$3000, 1/15, 84$	$\text{UT}_{2+\text{HOPoRTs}}$	56	66	681	131,797	1,581,574
$3000, 1/15, 84$	$\text{UT}_{2+\text{HOT}}$	16	66	681	479,403	5,752,840
$3000, 1/15, 84$	$\text{UT}_{\aleph 2+\text{HOT}}$	16	66	681	∞	∞

Table 18: Similar to the previous table, but with $k = 20$ kB/s instead of 3 kB/s.

k, B_f, B_h	Network	E. B_h (B)	E. D_h (B)	Scale $\times$	N_2	Σ TPS
$20000, 1/600, 80$	Bitcoin	80	-	1	-	80
$20000, 1/20, 1070$	Cardano	1070	-	1	-	80
$20000, 1.58, 144$	Solana	144	-	1	-	80
$20000, 1/15, 84$	$\text{UT}_{1+\text{PoRTs}}$	125	-	1	-	48,009
$20000, 1/15, 84$	$\text{UT}_{1+\text{HOPoRTs}}$	77	-	1	-	77,078
$20000, 1/15, 84$	$\text{UT}_{1+\text{HOT}}$	16	-	1	-	375,000
$20000, 1/6, 288$	Polkadot	288	819	146	146	11,721
$20000, 1/12, 200$	Eth2	200	312	769	769	61,538
$20000, 1/15, 66$	Opt.Shard	66	66	4,545	4,545	363,636
$20000, 1/15, 84$	$\text{UT}_{2+\text{PoRTs}}$	125	66	4,545	2,727,810	2.18×10^8
$20000, 1/15, 84$	$\text{UT}_{2+\text{HOPoRTs}}$	77	66	4,545	4,379,483	3.50×10^8
$20000, 1/15, 84$	$\text{UT}_{2+\text{HOT}}$	16	66	4,545	21,306,818	1.70×10^9
$20000, 1/15, 84$	$\text{UT}_{\aleph 2+\text{HOT}}$	16	66	4,545	∞	∞

Table 19: A comparison of computational requirements (approximated by k) for 1 million TPS between UT and various other networks. UT_2's equivalent TPS is also provided (given the same parameters of k, B_f and B_h).

k, B_f, B_h	Network	N_2	Σ TPS	Equiv. $\text{UT}_{2+\text{OPT}}$ Σ TPS
$2.5 \times 10^8, 1/600, 80$	Bitcoin	-	1,000,000	1.37×10^{24}
$2.5 \times 10^8, 1/20, 1070$	Cardano	-	1,000,000	5.80×10^{18}
$2.5 \times 10^8, 1.58, 144$	Solana	-	1,000,000	4.99×10^{17}
$93680, 1/15, 84$	$\text{UT}_{1+\text{PoRTs}}$	-	1,000,011	4.25×10^{10}
$74136, 1/15, 84$	$\text{UT}_{1+\text{HOPoRTs}}$	-	1,000,049	2.10×10^{10}
$32660, 1/15, 84$	$\text{UT}_{1+\text{HOT}}$	-	999,999	1.80×10^9
$184800, 1/6, 288$	Polkadot	1,353	1,000,763	3.47×10^9
$244200, 1/12, 200$	Eth2	1,024	1,000,243	5.03×10^{10}
$33166, 1/15, 66$	Opt.Shard	7,537	1,000,000	3.28×10^9
$3074, 1/15, 84$	$\text{UT}_{2+\text{PoRTs}}$	81,348	1,000,264	1,500,397
$2526, 1/15, 84$	$\text{UT}_{2+\text{HOPoRTs}}$	99,046	1,000,761	832,520
$1674, 1/15, 84$	$\text{UT}_{2+\text{HOT}}$	149,318	999,999	242,424

Table 20: Table evaluating various other networks against trilemma criteria.

Network	Decentralized?	$O(n)$ Secure?	$O(n)$ Scalable?	
Bitcoin	Yes	Yes	No	(Max. TPS: $\sim$ 5)[*]
Cardano	Yes	Maybe[PoS]	No	(Max. TPS: $\sim$ 10 - 80)[#]
Solana	No[‡]	No ($O(c)$)[†]	Unlikely	(Max. TPS: $\sim$ 50K)[‡]
Polkadot	Maybe	Maybe[PoS]	Unlikely	(Max. TPS: $\sim$ 200 - 12K)[¶]
Eth2	Maybe	Maybe[PoS]	Unlikely	(Max. TPS: $\sim$ 1K - 62K)[‖]
Opt.Shard	Yes	Yes	Maybe	(Max. TPS: $\sim$ 8K - 350K)[§]

[*] Real-world performance of Bitcoin; $k \approx 1700$ B/s; $\text{Tx}_{\text{avg}} \approx 375$ B.

[#,‖,¶] Prediction based on $3000 \leq k \leq 20000$.

[§] Assuming $3000 \leq k \leq 20000$ and that child-chains use PoW + PoR with the smallest possible headers.

[PoS] Unanswered criticisms of PoS (replies) mean that we cannot conclude that PoS is $O(n)$ secure. Saying PoS is “Maybe” $O(n)$ secure (especially when used in isolation) reflects an optimistic perspective — since there are unanswered criticisms, we should arguably conclude “No”. (There are no such unanswered criticisms of PoW-based consensus.)

[†] PoH is $O(c)$ secure by design. Solana uses PoS on top of PoH, but this is applied after-the-fact and it’s not clear that a well-placed and well-resourced attacker would require more than $O(c)$ resources. Additionally, since a $\leq O(c)$ DoS has brought down Solana before, this is an upper limit on Solana’s security.

[‡] Even though Solana requires $k \approx 10^7$ ($\sim$ 12 MB/s) for 50K TPS, this is consistent with their published “strategy” for “scaling”. 50K is chosen as the limit because the official site claims *at least* 50K as the upper limit and 400K is known to be beyond Solana’s capabilities. (Note: this also earns Solana a decisive “No” under the “Decentralized?” column.)

B UT Variant Complexities

B.1 +PoRs: Explicit PoRs

What is the throughput of the simplex if simplex-chains include explicit proofs of reflection (as merkle branches) and the headers of reflecting chains?

Let g be the length of the digest in bytes, i.e., the size of the hashes used in our merkle trees.

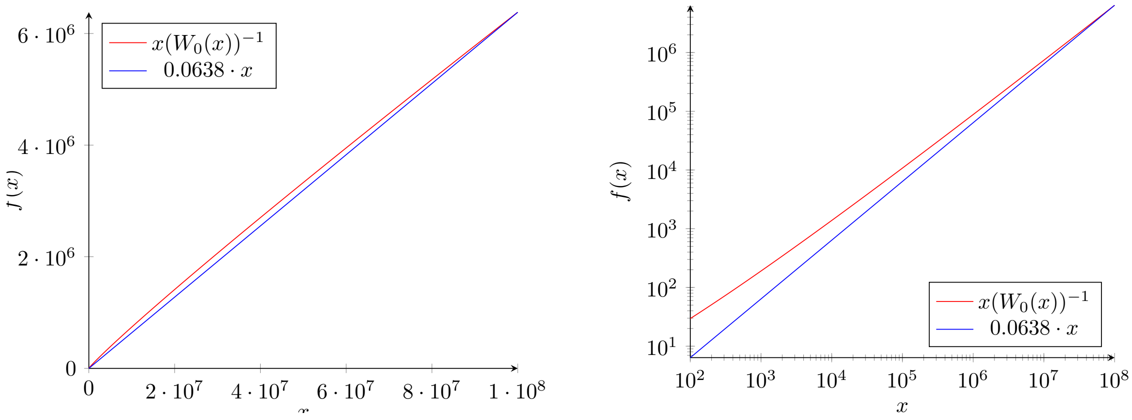
$$\begin{aligned}
k_1 &= k_{1,tx} + k_{1,B} \\
&= k_{1,tx} + B_f \cdot N_1 \cdot (B_h + g \cdot \lceil \log_2 N_1 \rceil) \\
&\approx k_{1,tx} + B_f \cdot N_1 \cdot (B_h + g \cdot \log_2 N_1) \\
\implies T_1 &= N_1 \cdot (k_1 - B_f \cdot N_1 \cdot (B_h + g \cdot \log_2 N_1)) \\
\frac{dT_1}{dN_1} &= \frac{1}{\ln 2}(k_1 \cdot \ln 2 - B_f \cdot N_1 \cdot (g + B_h \cdot \ln 4) - 2 \cdot B_f \cdot g \cdot N_1 \cdot \ln N_1) \qquad (63)
\end{aligned}$$

which has a zero at:

$$\begin{aligned}
N_1 &= \frac{k_1 \cdot \ln 2}{2 \cdot B_f \cdot g \cdot W_0((2^{B_h g^{-1} - 1} \cdot \sqrt{e} \cdot k_1 \cdot \ln 2)(B_f \cdot g)^{-1})} \\
\implies k_{1,tx} &\neq k_{1,B}
\end{aligned}$$

Note: $W_0(z)$ is the Lambert W function, a.k.a. the product logarithm.

Equation 63 gives an N_1 that is of the form $N_1 = O(1) \cdot \frac{k_1}{W_0(O(1) \cdot k_1)}$. Figure 39 graphs $f(x) = \frac{x}{W_0(x)}$ for values of x that we care about; $f(x) = 0.0638x$ is included for comparison. Regarding this specific case, approximating $O(\frac{k}{W_0(k)}) = O(k)$ gives us $O(N_1) = O(c)$ and $O(T_1) = O(c^2)$.



(a) Using linear axes (b) Using log axes

Figure 39: Graphs of $f(x) = \frac{x}{W_0(x)}$ for $x \in [10^2, 10^8]$.

If we use verkle PoRs instead of merkle PoRs, then we'll have a different $k_{1,B}$. Assuming that vector commitments and proofs are g bytes, and vector locations are 1 byte:

$$k_{1,B} = B_f \cdot N_1 \cdot (B_h + (1 + g) \cdot \log_{256} N_1) \qquad (64)$$

Other than this change, the logic that is used in Equation 63 still works, and the resulting complexities will be the same.

Table 21: UT_{+PoRs} capacity given different parameters.

k, B_f, B_h	N_1	Σ TPS_1	N_2	Σ TPS_2	PoR (B)	$\mathbb{C}'$ (Hz)
$3000, {}^1/_{7.5}, 112$	78	465	7,793	93,516	33	10.4
$3000, {}^1/_{7.5}, 84$	96	576	12,877	154,532	33	12.8
$3000, {}^1/_{15}, 112$	155	931	31,172	374,075	33	10.3
$3000, {}^1/_{15}, 84$	192	1,153	51,510	618,130	33	12.8
$3000, {}^1/_{30}, 112$	276	1,813	121,457	1,457,487	35	9.2
$3000, {}^1/_{30}, 84$	328	2,152	192,172	2,306,074	40	10.9
$3000, {}^1/_{60}, 112$	511	3,320	444,647	5,335,770	49	8.5
$3000, {}^1/_{60}, 84$	628	3,943	704,122	8,449,470	52	10.5
$20000, {}^1/_{7.5}, 112$	445	18,797	314,698	25,175,874	47	59.3
$20000, {}^1/_{7.5}, 84$	511	22,283	497,403	39,792,300	49	68.1
$20000, {}^1/_{15}, 112$	866	35,610	1,192,304	95,384,332	56	57.7
$20000, {}^1/_{15}, 84$	1,023	42,256	1,886,448	1.51×10^8	57	68.2
$20000, {}^1/_{30}, 112$	1,709	69,282	4,639,446	3.71×10^8	61	56.0
$20000, {}^1/_{30}, 84$	2,027	82,206	7,339,839	5.87×10^8	61	67.6
$20000, {}^1/_{60}, 112$	3,327	136,594	18,293,925	1.46×10^9	63	55.4
$20000, {}^1/_{60}, 84$	3,839	161,784	28,890,120	2.31×10^9	63	63.0
$184800, {}^1/_{15}, 112$	7,167	2,874,121	96,231,756	7.11×10^{10}	64	477.8
$244200, {}^1/_{15}, 112$	9,471	5,011,856	1.68×10^8	1.64×10^{11}	65	631.4

B.1.1 +PoRs with +T

Table 22: UT_{+PoRTs} capacity given different parameters.

k, B_f, B_h	N_1	Σ TPS_1	N_2	Σ TPS_2	PoR (B)	$\mathbb{C}'$ (Hz)
$3000, {}^1/_{7.5}, 112$	100	597	13,999	167,999	33	13.3
$3000, {}^1/_{7.5}, 84$	114	681	19,369	232,435	33	15.2
$3000, {}^1/_{15}, 112$	199	1,194	56,001	672,013	33	13.3
$3000, {}^1/_{15}, 84$	227	1,363	77,479	929,750	33	15.1
$3000, {}^1/_{30}, 112$	337	2,211	207,309	2,487,717	41	11.2
$3000, {}^1/_{30}, 84$	373	2,445	277,935	3,335,230	43	12.4
$3000, {}^1/_{60}, 112$	645	4,051	759,584	9,115,011	53	10.8
$3000, {}^1/_{60}, 84$	714	4,480	1,018,358	12,220,304	54	11.9
$20000, {}^1/_{7.5}, 112$	511	22,840	535,329	42,826,399	49	68.1
$20000, {}^1/_{7.5}, 84$	600	25,338	719,836	57,586,909	52	80.0
$20000, {}^1/_{15}, 112$	1,023	43,372	2,033,097	1.63×10^8	57	68.2
$20000, {}^1/_{15}, 84$	1,168	48,009	2,727,810	2.18×10^8	58	77.9
$20000, {}^1/_{30}, 112$	2,047	84,433	7,915,621	6.33×10^8	61	68.2
$20000, {}^1/_{30}, 84$	2,303	93,405	10,614,317	8.49×10^8	62	76.8
$20000, {}^1/_{60}, 112$	3,839	165,714	31,071,522	2.49×10^9	63	63.0
$20000, {}^1/_{60}, 84$	4,095	182,203	41,409,942	3.31×10^9	64	68.3
$184800, {}^1/_{15}, 112$	8,703	3,501,439	1.64×10^8	1.21×10^{11}	65	580.2
$244200, {}^1/_{15}, 112$	11,519	6,107,287	2.86×10^8	2.80×10^{11}	65	767.9

B.2 +OP and +OPT

The +OP variants exclude PoRs from simplex-chain blocks. This is reasonable if users running full nodes are willing to download and temporarily store all blocks from all simplex-chains (this allows each node to regenerate the PoRs). The PoRs are still processed as part of a chain's state transition (each reflecting header will have a corresponding PoR), and are thus provable. Additionally, since full nodes will need to recalculate these, a suitable P2P protocol will allow PoRs to be requested from full nodes on an ad-hoc basis.

The derivations of +OP's complexity was covered in Section 5.

Table 23: $\mathrm{UT}_{+\mathrm{OP}}$ capacity given different parameters.

k, B_f, B_h	N_1	Σ TPS$_1$	N_2	Σ TPS$_2$	PoR (B)	$\mathbb{C}'$ (Hz)
$3000, 1/7.5, 112$	100	602	10,089	121,073	33	13.4
$3000, 1/7.5, 84$	133	803	17,936	215,242	33	17.9
$3000, 1/15, 112$	200	1,205	40,357	484,295	33	13.4
$3000, 1/15, 84$	267	1,607	71,747	860,969	34	17.9
$3000, 1/30, 112$	401	2,410	161,431	1,937,181	45	13.4
$3000, 1/30, 84$	535	3,214	286,989	3,443,877	50	17.9
$3000, 1/60, 112$	803	4,821	645,727	7,748,724	55	13.4
$3000, 1/60, 84$	1,071	6,428	1,147,959	13,775,510	58	17.9
$20000, 1/7.5, 112$	669	26,785	448,421	35,873,724	53	89.3
$20000, 1/7.5, 84$	892	35,714	797,193	63,775,510	56	119.0
$20000, 1/15, 112$	1,339	53,571	1,793,686	1.43×10^8	59	89.3
$20000, 1/15, 84$	1,785	71,428	3,188,775	2.55×10^8	61	119.0
$20000, 1/30, 112$	2,678	107,142	7,174,744	5.74×10^8	62	89.3
$20000, 1/30, 84$	3,571	142,857	12,755,102	1.02×10^9	63	119.0
$20000, 1/60, 112$	5,357	214,285	28,698,979	2.30×10^9	64	89.3
$20000, 1/60, 84$	7,142	285,714	51,020,408	4.08×10^9	64	119.0
$184800, 1/15, 112$	12,375	4,573,800	1.53×10^8	1.13×10^{11}	65	825.0
$244200, 1/15, 112$	16,352	7,986,648	2.67×10^8	2.61×10^{11}	65	1,090.2

Table 24: $\mathrm{UT}_{+\mathrm{OPT}}$ capacity given different parameters.

k, B_f, B_h	N_1	Σ TPS$_1$	N_2	Σ TPS$_2$	PoR (B)	$\mathbb{C}'$ (Hz)
$3000, 1/7.5, 112$	140	843	19,775	237,304	33	18.8
$3000, 1/7.5, 84$	170	1,022	29,054	348,657	33	22.7
$3000, 1/15, 112$	281	1,687	79,101	949,218	36	18.8
$3000, 1/15, 84$	340	2,045	116,219	1,394,628	41	22.7
$3000, 1/30, 112$	562	3,375	316,406	3,796,875	51	18.8
$3000, 1/30, 84$	681	4,090	464,876	5,578,512	53	22.7
$3000, 1/60, 112$	1,125	6,750	1,265,625	15,187,500	58	18.8
$3000, 1/60, 84$	1,363	8,181	1,859,504	22,314,049	59	22.7
$20000, 1/7.5, 112$	937	37,500	878,906	70,312,500	57	125.0
$20000, 1/7.5, 84$	1,136	45,454	1,291,322	1.03×10^8	58	151.5
$20000, 1/15, 112$	1,875	75,000	3,515,625	2.81×10^8	61	125.0
$20000, 1/15, 84$	2,272	90,909	5,165,289	4.13×10^8	62	151.5
$20000, 1/30, 112$	3,750	150,000	14,062,500	1.13×10^9	63	125.0
$20000, 1/30, 84$	4,545	181,818	20,661,157	1.65×10^9	64	151.5
$20000, 1/60, 112$	7,500	300,000	56,250,000	4.50×10^9	65	125.0
$20000, 1/60, 84$	9,090	363,636	82,644,628	6.61×10^9	65	151.5
$184800, 1/15, 112$	17,325	6,403,320	3.00×10^8	2.22×10^{11}	65	1,155.0
$244200, 1/15, 112$	22,893	11,181,307	5.24×10^8	5.12×10^{11}	65	1,526.3

B.3 +HO and +HOT

The +HO variants replace the headers of reflecting chains with the respective header's hash. This is reasonable since the headers of each simplex-chain (that would otherwise be recorded in simplex-chain blocks) are *common* among all simplex-chains. If a user is running nodes for multiple simplex-chains, they should only need to download each header once — including raw headers in each block is redundant.

Thus, +HO has $k_{1,B}$ of:

$$k_{1,B} = N_1 \cdot B_f \cdot g \tag{65}$$

This is equivalent to +OP with very small headers — 32 bytes instead of 80+ bytes.

Table 25: $\text{UT}_{+\text{HO}}$ capacity given different parameters.

k, B_f, B_h	N_1	Σ TPS$_1$	N_2	Σ TPS$_2$	PoR (B)	$\mathbb{C}'$ (Hz)
$3000, 1/7.5, 112$	351	2,109	35,313	423,758	42	46.9
$3000, 1/7.5, 84$	351	2,109	47,084	565,011	42	46.9
$3000, 1/15, 112$	703	4,218	141,252	1,695,033	54	46.9
$3000, 1/15, 84$	703	4,218	188,337	2,260,044	54	46.9
$3000, 1/30, 112$	1,406	8,437	565,011	6,780,133	60	46.9
$3000, 1/30, 84$	1,406	8,437	753,348	9,040,178	60	46.9
$3000, 1/60, 112$	2,812	16,875	2,260,044	27,120,535	63	46.9
$3000, 1/60, 84$	2,812	16,875	3,013,392	36,160,714	63	46.9
$20000, 1/7.5, 112$	2,343	93,750	1,569,475	1.26×10^8	62	312.5
$20000, 1/7.5, 84$	2,343	93,750	2,092,633	1.67×10^8	62	312.5
$20000, 1/15, 112$	4,687	187,500	6,277,901	5.02×10^8	64	312.5
$20000, 1/15, 84$	4,687	187,500	8,370,535	6.70×10^8	64	312.5
$20000, 1/30, 112$	9,375	375,000	25,111,607	2.01×10^9	65	312.5
$20000, 1/30, 84$	9,375	375,000	33,482,142	2.68×10^9	65	312.5
$20000, 1/60, 112$	18,750	750,000	1.00×10^8	8.04×10^9	65	312.5
$20000, 1/60, 84$	18,750	750,000	1.34×10^8	1.07×10^{10}	65	312.5
$184800, 1/15, 112$	43,312	16,008,300	5.36×10^8	3.96×10^{11}	65	2,887.5
$244200, 1/15, 112$	57,234	27,953,268	9.36×10^8	9.14×10^{11}	65	3,815.6

Table 26: $\text{UT}_{+\text{HOT}}$ capacity given different parameters.

k, B_f, B_h	N_1	Σ TPS$_1$	N_2	Σ TPS$_2$	PoR (B)	$\mathbb{C}'$ (Hz)
$3000, 1/7.5, 112$	703	4,218	98,876	1,186,523	54	93.8
$3000, 1/7.5, 84$	703	4,218	119,850	1,438,210	54	93.8
$3000, 1/15, 112$	1,406	8,437	395,507	4,746,093	60	93.8
$3000, 1/15, 84$	1,406	8,437	479,403	5,752,840	60	93.8
$3000, 1/30, 112$	2,812	16,875	1,582,031	18,984,375	63	93.8
$3000, 1/30, 84$	2,812	16,875	1,917,613	23,011,363	63	93.8
$3000, 1/60, 112$	5,625	33,750	6,328,125	75,937,500	64	93.8
$3000, 1/60, 84$	5,625	33,750	7,670,454	92,045,454	64	93.8
$20000, 1/7.5, 112$	4,687	187,500	4,394,531	3.52×10^8	64	625.0
$20000, 1/7.5, 84$	4,687	187,500	5,326,704	4.26×10^8	64	625.0
$20000, 1/15, 112$	9,375	375,000	17,578,125	1.41×10^9	65	625.0
$20000, 1/15, 84$	9,375	375,000	21,306,818	1.70×10^9	65	625.0
$20000, 1/30, 112$	18,750	750,000	70,312,500	5.63×10^9	65	625.0
$20000, 1/30, 84$	18,750	750,000	85,227,272	6.82×10^9	65	625.0
$20000, 1/60, 112$	37,500	1,500,000	2.81×10^8	2.25×10^{10}	65	625.0
$20000, 1/60, 84$	37,500	1,500,000	3.41×10^8	2.73×10^{10}	65	625.0
$184800, 1/15, 112$	86,625	32,016,600	1.50×10^9	1.11×10^{12}	74	5,775.0
$244200, 1/15, 112$	114,468	55,906,537	2.62×10^9	2.56×10^{12}	80	7,631.3

B.4 +HOPoRs and +HOPoRTs

+HOPoRs is the combination of +HO and +PoRs — headers are omitted but PoRs are still explicitly recorded.

Thus, +HOPoRs has $k_{1,B}$ of:

$$\begin{aligned} \text{Merkle PoRs:} \quad & k_{1,B} = B_f \cdot N_1 \cdot g \cdot (1 + \log_2 N_1) \\ \text{Verkle PoRs:} \quad & k_{1,B} = B_f \cdot N_1 \cdot (g + (g+1) \cdot \max(1, \log_{256} N_1)) \end{aligned} \tag{66}$$

Table 27: $\text{UT}_{\text{+HOPoRs}}$ capacity given different parameters.

k, B_f, B_h	N_1	Σ TPS$_1$	N_2	Σ TPS$_2$	PoR (B)	$\mathbb{C}'$ (Hz)
$3000, 1/7.5, 112$	173	1,038	17,384	208,619	33	23.1
$3000, 1/7.5, 84$	173	1,038	23,179	278,159	33	23.1
$3000, 1/15, 112$	273	1,940	64,986	779,834	35	18.2
$3000, 1/15, 84$	273	1,940	86,648	1,039,779	35	18.2
$3000, 1/30, 112$	502	3,294	220,606	2,647,279	49	16.7
$3000, 1/30, 84$	502	3,294	294,142	3,529,705	49	16.7
$3000, 1/60, 112$	961	6,033	808,020	9,696,249	57	16.0
$3000, 1/60, 84$	961	6,033	1,077,361	12,928,333	57	16.0
$20000, 1/7.5, 112$	767	34,040	569,881	45,590,480	55	102.3
$20000, 1/7.5, 84$	767	34,040	759,841	60,787,306	55	102.3
$20000, 1/15, 112$	1,535	64,614	2,163,431	1.73×10^8	60	102.3
$20000, 1/15, 84$	1,535	64,614	2,884,575	2.31×10^8	60	102.3
$20000, 1/30, 112$	2,815	124,703	8,350,697	6.68×10^8	63	93.8
$20000, 1/30, 84$	2,815	124,703	11,134,263	8.91×10^8	63	93.8
$20000, 1/60, 112$	5,375	244,038	32,683,743	2.61×10^9	64	89.6
$20000, 1/60, 84$	5,375	244,038	43,578,324	3.49×10^9	64	89.6
$184800, 1/15, 112$	12,287	5,159,789	1.73×10^8	1.28×10^{11}	65	819.1
$244200, 1/15, 112$	16,127	8,983,624	3.01×10^8	2.94×10^{11}	65	1,075.1

Table 28: $\text{UT}_{\text{+HOPoRTs}}$ capacity given different parameters.

k, B_f, B_h	N_1	Σ TPS$_1$	N_2	Σ TPS$_2$	PoR (B)	$\mathbb{C}'$ (Hz)
$3000, 1/7.5, 112$	230	1,377	32,286	387,434	33	30.7
$3000, 1/7.5, 84$	230	1,377	39,134	469,618	33	30.7
$3000, 1/15, 112$	326	2,319	108,733	1,304,798	40	21.7
$3000, 1/15, 84$	326	2,319	131,797	1,581,574	40	21.7
$3000, 1/30, 112$	511	3,850	360,976	4,331,721	49	17.0
$3000, 1/30, 84$	511	3,850	437,547	5,250,571	49	17.0
$3000, 1/60, 112$	1,023	7,124	1,335,820	16,029,847	57	17.1
$3000, 1/60, 84$	1,023	7,124	1,619,176	19,430,117	57	17.1
$20000, 1/7.5, 112$	966	40,771	955,579	76,446,341	57	128.8
$20000, 1/7.5, 84$	966	40,771	1,158,277	92,662,232	57	128.8
$20000, 1/15, 112$	1,791	77,078	3,613,073	2.89×10^8	61	119.4
$20000, 1/15, 84$	1,791	77,078	4,379,483	3.50×10^8	61	119.4
$20000, 1/30, 112$	3,327	148,711	13,941,710	1.12×10^9	63	110.9
$20000, 1/30, 84$	3,327	148,711	16,899,042	1.35×10^9	63	110.9
$20000, 1/60, 112$	6,143	288,296	54,055,558	4.32×10^9	64	102.4
$20000, 1/60, 84$	6,143	288,296	65,521,889	5.24×10^9	64	102.4
$184800, 1/15, 112$	14,335	6,128,357	2.87×10^8	2.12×10^{11}	65	955.7
$244200, 1/15, 112$	18,687	10,647,792	4.99×10^8	4.88×10^{11}	65	1,245.8

C CEC Experiment Extra

C.1 Results with $q = 0.48$

Figure 40 shows results of the simulation for $q = 0.48$. Unlike earlier figures, this figure uses $c = 20$ as the basis for comparison instead of $c = 5$ to accommodate the extended duration of attacks performed by an attacker close to 50% of the global hash-rate. These results do converge as predicted by the CEC, but not as closely as the results in Section 4.10.4.

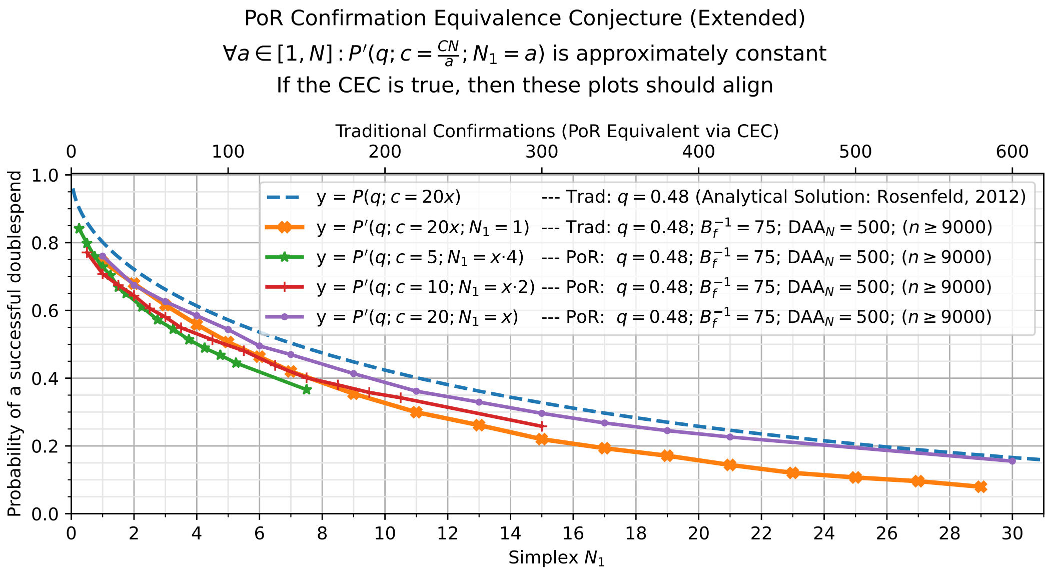


Figure 40: Main results for $q = 0.48$ for the experiment documented in Section 4.10.4

Blockchain security breaks down at $q \geq 0.5$, so, as $q \to 0.5$ we should expect that *previously insignificant* implementation details become more and more significant. Any previously insignificant advantages for the honest network or the attacker (at lower values of q) can become substantial as q approaches that threshold. It might be possible to further refine the simulation (via additional iterations and error corrections), but at some point it's not worth it.

So we should expect the simulation to lose accuracy as $q \to 0.5$. Given this, it seems reasonable to conclude that Figure 40 is consistent with the CEC's predictions, albeit with the caveat of lower accuracy.

C.2 Simulator Iteration

The purpose of this section is to avoid excluding all but the best results, and to document the steps taken that increased the accuracy of the simulation.

Other than the major changes discussed in Section C.2.1 and on, many other configurations were also tested to establish that certain implementation details were *not* significant. These include:

- Hashing algorithm: `xxh3` was compared against `blake3` and `sha256`. No significant differences were observed.
- ${B_f}^{-1}$: No significant difference was found for values of 50, 75, and 100. The final value of 75 was, in part, chosen to ensure there was sufficient excess capacity in ${B_f}^{-1}$ (so that it did not affect results). Lower values (e.g., 10) seemed to work okay during early testing, but also result in more blocks being generated simultaneously.
- H: this value is important because $q \cdot H$ should result in a whole number. If H is too small, or $q \cdot H$ is not a whole number, loss of precision occurs. $H = 50$ is sufficient to test values of

q that are multiples of 0.02. $H = 75$ works for values of q that are multiples of 0.04, and was chosen over $H = 50$ to ensure excess capacity.

- Iteration between H and ${B_f}^{-1}$: the difficulty is the number of hashes to find a block, which, before the attack starts, is $\sim H \cdot {B_f}^{-1}$. Thus, $H = {B_f}^{-1} = 75$ is about *half* the computation cost of using $H = {B_f}^{-1} = 100$. Smaller values for both could work, but risks invalidating results if either is too small.

C.2.1 Initial Results

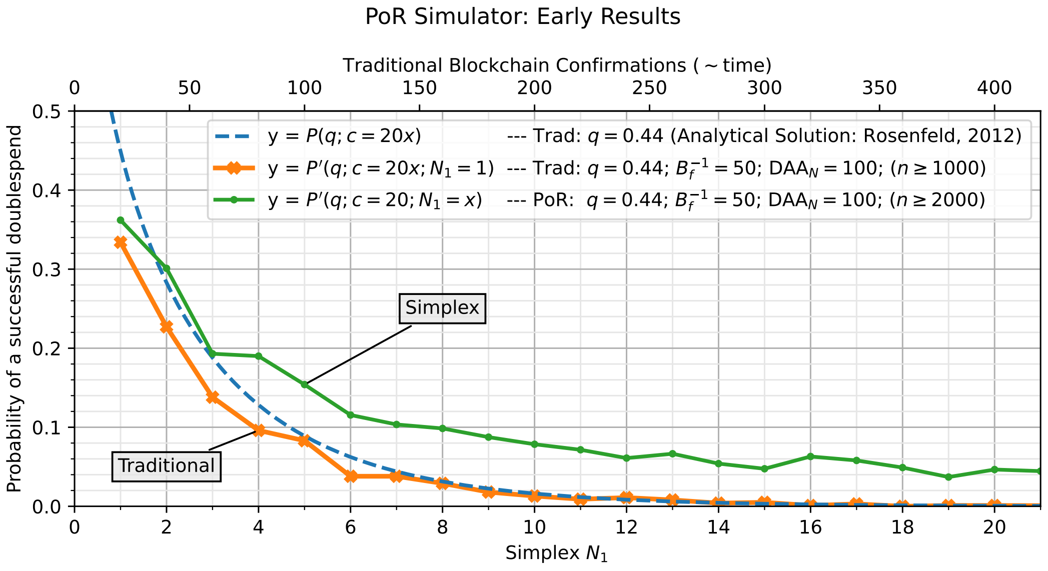


Figure 41: Early results showed that, although $P'(q, c = Cx, N_1 = 1) \to 0$ as expected (traditional chains), $P'(q, c = C, N_1 = x)$ did not approach 0 as N_1 increased (simplexes). This was prior to the substantive error corrections shown in subsequent figures.

C.2.2 Accounting for Draft Reflecting Work

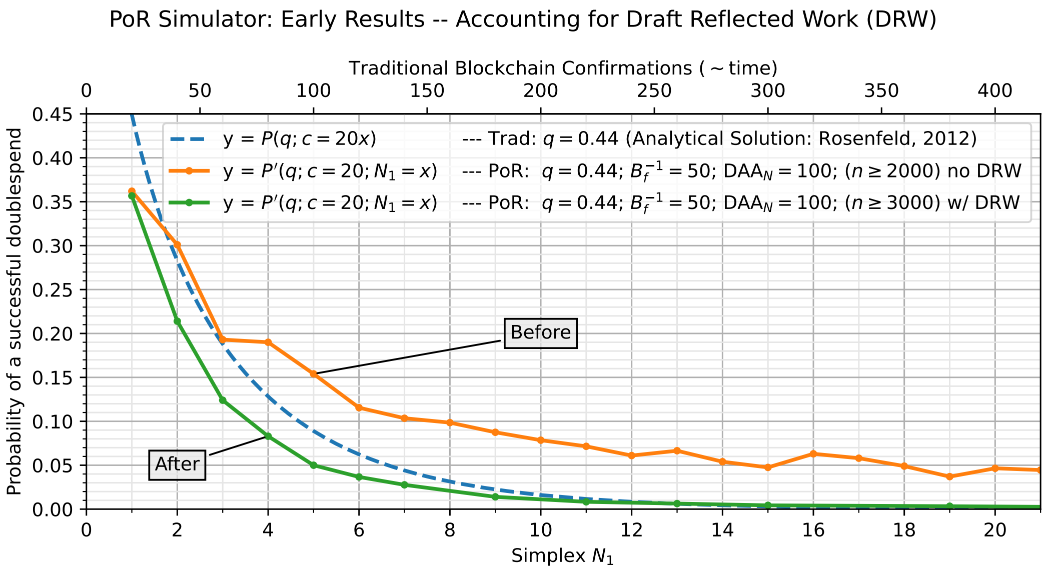


Figure 42: Early results (legend: 'no DRW') did not show $P' \to 0$ as N_1 increased. This was fixed after accounting for draft reflected work ('w/ DRW') in the function that determines whether an attack has succeeded. The significance of this is explained in Section 4.10.3.

C.2.3 Attacker's Hash-Rate Distribution: Uniform vs Random

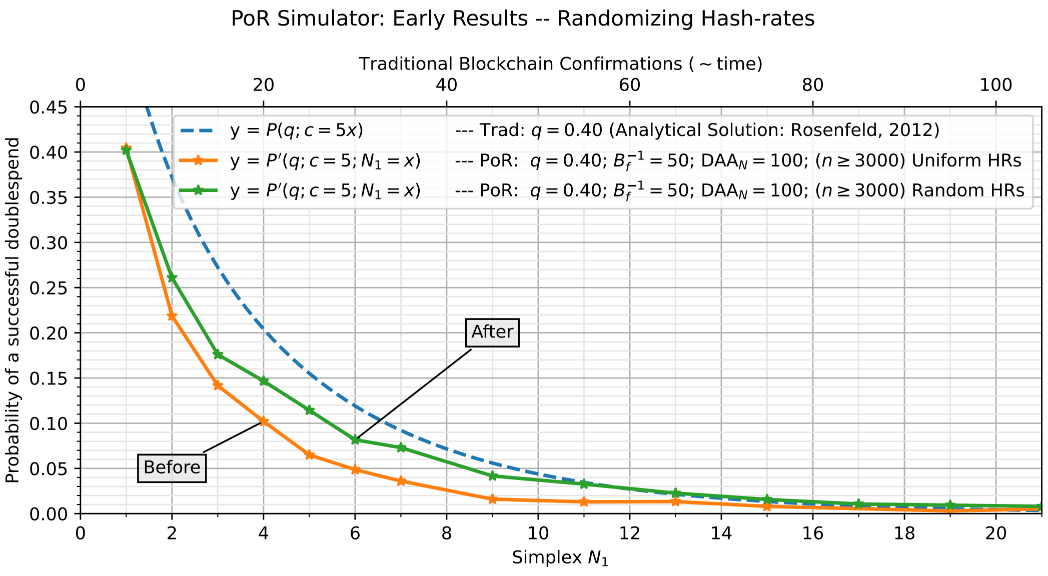


Figure 43: These results indicate that simplexes are *more* secure when an attacker has the same hash-rate over all simplex-chains. Randomizing hash-rates (of both the honest network and the attacker) over all simplex-chains, while globally maintaining $p + q = 1$, improves the accuracy of the simulation with respect to the CEC's predictions.

When hash-rates are *uniformly* distributed: the miners of each chain do H hashes per tick, and the attacker does $q \cdot H$ of those hashes per tick.

When hash-rates are *randomly* distributed: for all chains but the primary chain, $H \cdot (N_1 - 1)$ hashes per tick are distributed randomly over those chains, with the attacker responsible for $q \cdot H \cdot (N_1 - 1)$ of those hashes per tick. Some chains will have a hash-rate greater than H hashes per tick, and some chains fewer. It's possible for an attacker to have *any* proportion of a particular chain's hash-rate, but $p + q = 1$ is maintained globally. Both the attacker's and honest miners' hash-rates are uniformly distributed, and thus the hash-rate of each chain has a trapezoidal distribution. PMFs of the hash-rate of each chain, the honest miners, and the attacker can be obtained via `make print-randhr-pmfs`.

C.2.4 The Attacker's "Bonus Block"

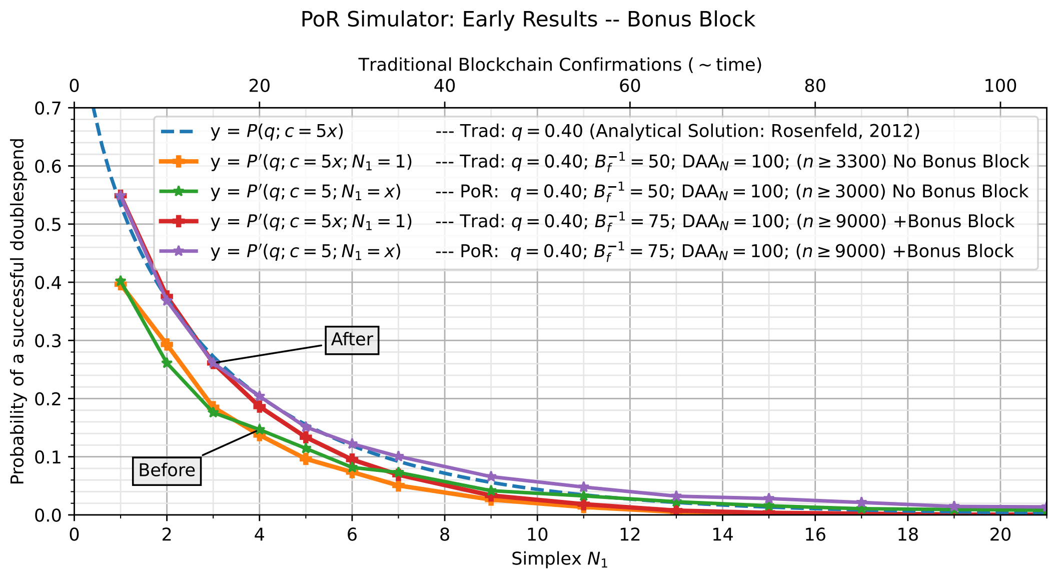


Figure 44: In general, an attacker has the opportunity to prepare for a doublespend attack. Particularly, they can wait to mine their first private block before making the public transaction (which is the target of the doublespend). When the attacker waits to mine this first private block before starting the attack, it's called (in this paper) the "bonus block". Implementing this substantially increased the accuracy of the simulation with respect to the CEC's predictions for smaller values of c and/or N_1.

C.2.5 Effects of DAA$_N$

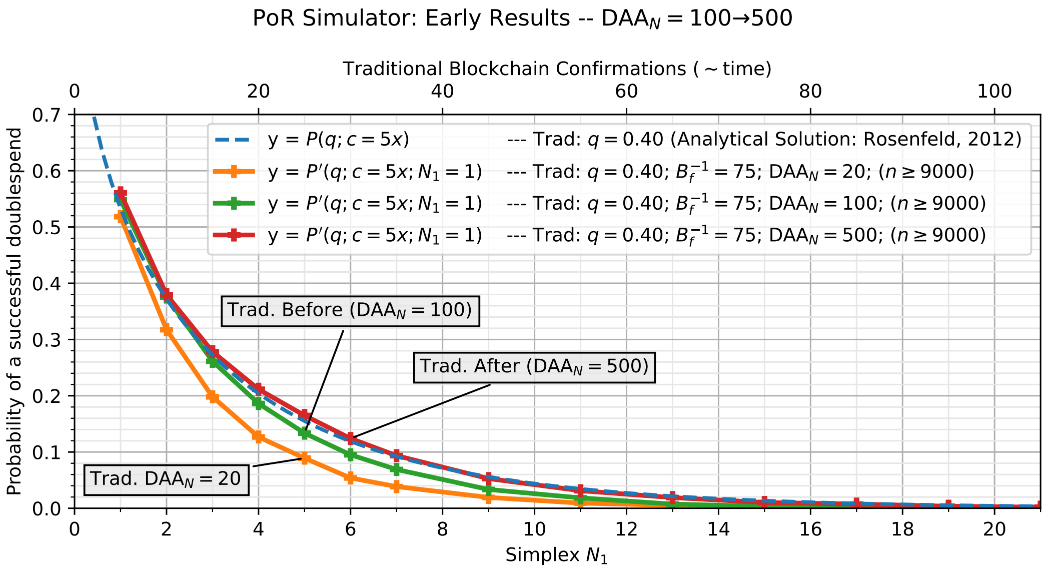


Figure 45: This was the last iteration prior to generating the final results (Section 4.10.4.5). Early results indicated that lower values of DAA$_N$ were more secure for traditional chains, and that DAA$_N \approx 500$ was sufficient for results to closely match the analytical solution. DAA$_N$ is not as significant for the security of simplexes and PoR. Results for DAA$_N = 20$ are included to better demonstrate the effect of increasing DAA$_N$.