Blockclique: scaling blockchains through transaction sharding in a multithreaded block graph

Sébastien Forestier
contact@sforestier.com

Damir Vodenicarevic
damipator@gmail.com

Abstract

Decentralized crypto-currencies based on the blockchain architecture cannot scale to thousands of transactions per second. We design a new architecture, called the blockclique, combining transaction sharding, where transactions are separated into multiple groups based on their input address, and a multithreaded directed acyclic block graph structure, where each block references one previous block of each thread. Our open-source simulations show that in a network with an average upload bandwidth of 32 Mbps, the blockclique reaches a throughput of 6,000 transactions per second, with an average transaction time of 2 minutes.

1 Introduction

A **decentralized crypto-currency** is a currency relying upon an open peer-to-peer network of computers sharing and executing transactions, usually in the form of a **blockchain**, where the use of cryptographic functions allows to prevent the theft or duplication of digital coins. Bitcoin, launched on January 3rd, 2009, was the first decentralized crypto-currency to appear [1], and the blockchain ecosystem developed over the years into a large set of crypto-currencies with very different properties. The advantages of decentralized crypto-currencies over traditional currencies are their relative anonymity, the relative speed of their transactions and the fact that no central authority could control or interfere with coin minting. Some crypto-currencies also allow to write data and smart contracts on the blockchain (e.g. Ethereum [2]), so that complex conditions involving multiple agents can be processed.

In a **decentralized** network, any computer (called node) can join the network at any time and download from the other peers the set of transactions that were executed up to this point. Any node should also be able to execute new transactions: to check that a transaction is valid (that money spent is money that has been received) and append it to a data structure, based on a crypto-currency protocol. However, if nodes execute too many valid transactions, they may not all be able to download and verify them, and reach a consensus about which transactions are considered executed. The protocol could thus limit the number of transactions that each node is allowed to execute and broadcast, but then a malicious agent could create many nodes that join the network in order to broadcast much more transactions to perturbate the network, which would be called a Sybil attack.

To regulate the rate of executed transactions, a decentralized crypto-currency protocol must perform a Sybil-resistant **random selection** of nodes that will execute transactions in a timely manner. For instance, Bitcoin and many other blockchains use a Proof-of-Work (PoW) mining scheme where nodes perform useless computational operations (called mining) until they find by chance a number that makes the cryptographic hash of a block of transactions be below a given target. A PoW mechanism that adapts the target to the current total mining power of the network ensures that random nodes create blocks with a constant average rate, each with a probability depending on its computational power. PoW is Sybil-resistant as if an agent divides its mining power between several nodes, its total mining power and thus its probability to create a block stays the same. Another random selection mechanism implemented in several protocols is Proof-of-Stake (PoS), see e.g. Cardano [3] and Tezos [4]. With PoS, nodes are randomly selected based on the amount of coins (called stake) that their addresses hold, to produce a block of transactions at a given time. For the different nodes to agree on the selection outcome, a pseudo-random mechanism is usually seeded from different sources such as the hash of the previous block and other inputs that are included in blocks from other nodes. PoS is also Sybil-resistant as the probability to be chosen is proportional to the stake, and has the advantage over PoW that nodes do not need to waste electricity in a mining power competition to get block rewards.

When a node is chosen, it executes transactions by appending them to a local **data structure** defined by the protocol. In the blockchain architecture, transactions are grouped together in blocks where each block contains the hash of a previous block. The data struc-

ture representing transactions is thus a block tree, with one genesis block that has no parent and multiple branches of blocks where each block has exactly one parent block.

Once the transaction data is created, the chosen node then broadcasts it to other nodes of the peer-to-peer network. However, as there is some communication latency between any two peers of the network, the different peers may not have received exactly the same set of transactions at any point in time, so that two nodes may broadcast incompatible transactions even if they are honest. The protocol thus specifies a **consensus rule** so that all peers can still agree about the set of transactions that are considered executed. In the blockchain architecture, if two nodes broadcast a different block with the same parent in the block tree, the Nakamoto consensus rule specifies that only the transactions in the block chain with the most work (the heaviest chain, called the blockchain) should be considered executed. This creates a fork of two alternative blockchains, and the nodes will work on one of the alternatives so that in the end only one of the two branches of the fork will ultimately become the blockchain.

A decentralized crypto-currency protocol thus defines three main components: a Sybil-resistant time-regulated random node selection mechanism (who appends data and when), a data structure (how data is appended), and a consensus rule (how to cope with incompatible data).

However, decentralized crypto-currencies based on the blockchain architecture face the problem of **scalability**. For instance, the Bitcoin network is limited to about 5 transactions per second (Ethereum to about 15), such that the transaction throughput is often saturated, which makes people spend large fees so that their transaction gets processed before others. Indeed, a queue of about $200,000$ transactions was waiting to be processed in the Bitcoin network in December 2017 leading to an average transaction fee of about 50 USD.

The trivial ways of increasing the **transaction throughput** would be to increase the number of transactions per block or the block frequency, however this is possible only to a small extent. With a small block size and a low block frequency (1 MB every 10 minutes on average in Bitcoin), the blocks have the time to be broadcasted to the whole network before another node creates a new block so the fork rate is low and the consensus is easy. However if we increase the block size too much (say to 1 GB) with the same block frequency (10 minutes on average), then the blocks would need much more time to be broadcasted and the fork rate would be too high, and if we increase the block frequency too much (say one block per second) with the same block size (1 MB), then again nodes would have the time to create many blocks before any of them could be broadcasted, therefore the fork rate would be too high and the network could not reach a stable consensus.

Recently, there have been several attempts to scale blockchains through changes in the data structure and the consensus rule. One line of work studies **transaction sharding** [5, 6, 7], a way to split the network of nodes and transactions into several groups so that transactions are processed in parallel, but in the end they only allow one chain of blocks on which the nodes of different groups have to agree upon. Another line of work tries to extend the block tree structure to a **block graph** structure [8, 9, 10], but they do not shard transactions so they need complex strategies to resolve incompatibilities between transactions of different blocks of the graph. Moreover, in those different lines of work, there has been no simulation regarding the crucial question of which transaction throughput can be reached depending on the number of nodes and the average bandwidth and latency in the network.

In this paper we describe a new architecture, called the **blockclique**, which is the first to combine transaction sharding and a directed acyclic block graph structure. The blockclique architecture separates blocks of transactions into multiple threads based on the transactions' input address, which allows to parallelize block production while preventing nodes from spending the same coins in several threads at the same time. Each block references one parent block in each thread, which builds a multithreaded directed acyclic block graph. The blockclique consensus rule defines the best clique of compatible blocks, which allows a block produced in a thread to be compatible with a block produced in another thread at the same time, and this way scales the rate of compatible blocks and transactions that can be broadcasted to the network while keeping the fork rate low. With respect to the blockchain, the blockclique architecture thus extends the data structure and adapts the Nakamoto consensus rule, however it is compatible with any Sybil-resistant time-regulated random selection mechanism, such as Proof-of-Work or Proof-of-Stake.

We provide an implementation of the blockclique data structure and consensus rule which allows to simulate the interaction of nodes in a peer-to-peer network that produce and broadcast blocks and compute the best clique of compatible blocks. Our **open-source simulations** show that if we use 32 threads with an average of 32 seconds between two blocks in each thread and a maximum block size of 1 MB in a PoW setting, then a throughput of more than $6,000$ transactions per second with a 2-minute transaction time is achieved in a network of nodes with an average upload bandwidth of 32 Mbps, with a low fork rate and with high security parameters. Furthermore, as there is on average one block per second added to the blockclique, the rewards are much more frequent than in the Bitcoin network for instance, such that in a PoW setting miners don't need to pool anymore which protects decentralization.

2 Related Work

The idea of sharding transactions into multiple groups appears in Elastico [5] and OmniLedger [6], however our approach to setup sharding is different. In both Elastico and OmniLedger, a Proof-of-Work challenge allows miners to generate an identity which assigns them to a random shard (or committee). The committees then use a PBFT (Practical byzantine fault tolerance) algorithm to agree on a set of transactions to commit from their shard, and finally random members of the committees agree to merge those different sets of transactions. However those are proof-of-concept and suffer limitations. For instance, in Elastico, there can still be consensus failures, and the consensus process is slow and thus transactions confirmation is slow. In Omniledger, the advantage of sharding is much reduced when transactions touch multiple shards (see [6]: Appendix C), which should however be considered as the default behavior. In the blockclique, transactions can have multiple input addresses from a same shard (called thread) and multiple outputs to addresses of any shard without interfering with consensus efficiency.

In Bitcoin-NG [11], a leader is elected once in a while through a Proof-of-Work scheme, who then decides which transactions to include in the blockchain. However, without sharding, their simulations do not show any scaling to high throughput, however they show that their architecture is fairer than standard blockchain protocols. Zilliqa [7] builds upon Bitcoin-NG with the election of leaders, and adds sharding. However they do not show any simulation results about the transaction throughput depending on the average network bandwidth and latency.

In those previous works, blocks are always structured in a block tree. Another recent line of work changes the way blocks are chained together. In the GHOST protocol [12], which is implemented in Ethereum, blocks point to their parent, but can also reference a list of stale uncle blocks that are not on the main chain, and that would have been forgotten otherwise. The advantage of this procedure is that stale blocks can be blocks from honest nodes, and here they can be taken into account when considering the total mining power of honest nodes, so that the mining power proportion of a group of attackers is not increased due to a potentially high stale rate.

The first directed acyclic block graph structure appears in [8] and SPECTRE [9]. In this work, blocks are ordered in a block DAG but transactions of one block can be incompatible with other transactions from other blocks because the transactions are not sharded. Thus a voting mechanism takes place to order transactions from block topology and decide which transactions are valid, however authors argue that the confirmation of transactions is fast. PHANTOM [10] is an extension of this work showing that if the communication delay diameter is assumed to be bounded by a constant, then a linear order between blocks can be found and the system has better liveness properties than SPECTRE. With the blockclique, we assume no bound on communication time, and shard transactions so that valid transactions of different threads are compatible by construction. The authors of SPECTRE and PHANTOM do not show either any simulation estimating the transaction throughput depending on the average network connectivity.

In IOTA [13], transactions are included in a directed acyclic graph of transactions (called the tangle), which is not sharded. When a user wants to create a transaction, it attaches its new transaction to two other tip transactions of its local transaction DAG, solves a small PoW scheme and broadcast the transaction. Consequently, there are no miners, no blocks and no Sybil-resistant time-regulation mechanism. The authors provide no simulations studying the transaction throughput, the computational cost of handling this transaction DAG, nor the transaction confirmation time depending on the average network connectivity. Moreover, a centralized node (called coordinator) run by the IOTA foundation provides checkpoints every minute so that all other nodes verify transactions in a same direction in the tangle and that the DAG do not grow too much in width. With the blockclique, the width of the block DAG is fixed by the number of threads, which naturally constrains the DAG to grow in one direction through multiple threads.

Finally, other very different approaches attempt to scale transaction throughput. One of them is to build multiple side chains where transactions can be included while not filling main chain's blocks (e.g. see [14]). However this leads to problems of high latency between side chains. Another one is to simply not write transactions on the main chain neither on side chains but off-chain, through micro-payment channels [15]. However in both approaches, as transactions are not written on the main chain, their security guarantees are not the same as the one of an immutable public ledger. Some other approaches try to scale architectures for consortium or centralized blockchains, within a permissioned group of users, see e.g. Ripple [16], Hashgraph [17], or [18], but we are only interested in decentralized networks here.

3 Blockclique

In this section, we first describe the blockclique architecture with its multithreaded directed acyclic block graph structure and the corresponding blockclique consensus rule. Second, we provide an example implementation of the transactions, blocks and rewards in a Proof-of-Work setting. Then, we study the security of the blockclique architecture against different types of attacks. Finally, we analyze the need for computing, network and storage resources.

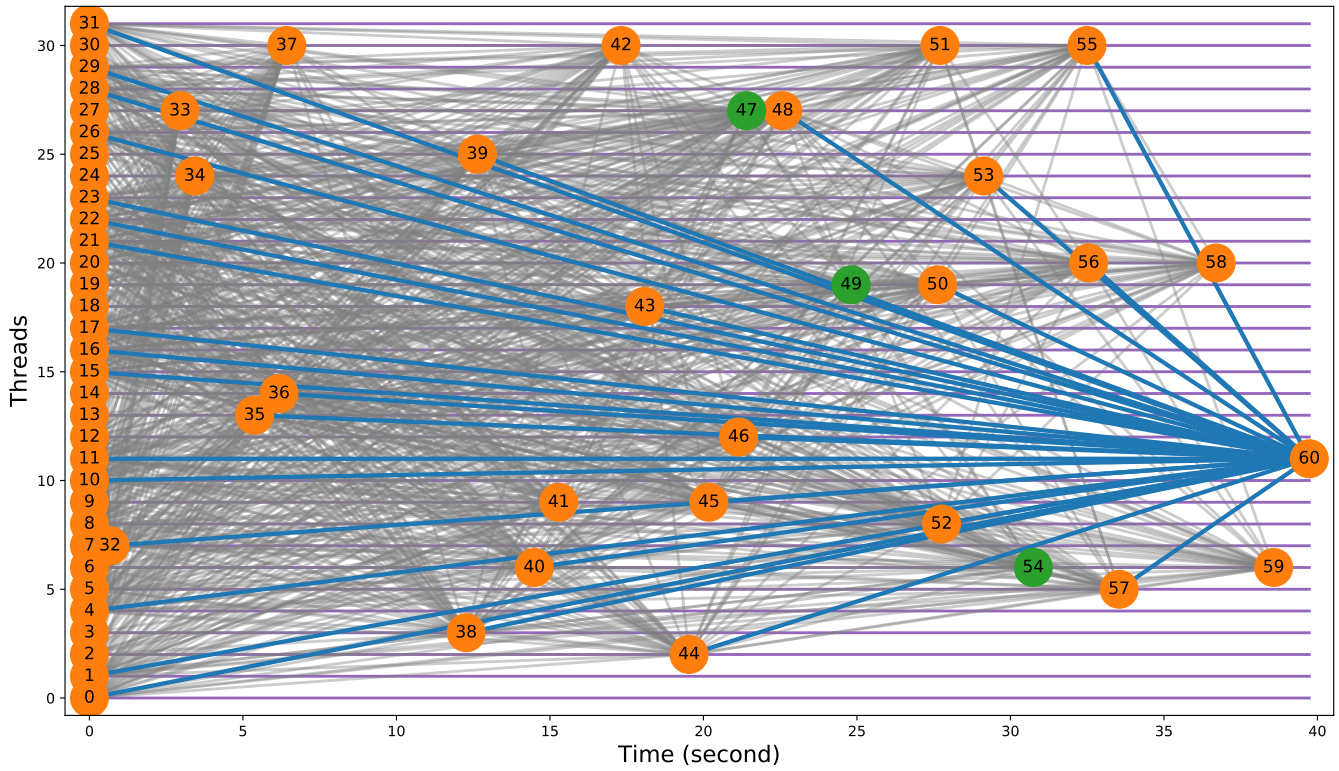


Figure 1: Example of a multithreaded directed acyclic block graph, with 32 parallel threads and one block created every 32s on average in each thread. The x axis shows time and the y axis shows the 32 threads. We plot one orange circle per block currently in the blockclique, on its corresponding thread and at the time it was created, and we plot a green circle for blocks that are not in the blockclique. The first 32 blocks correspond to the genesis blocks which are created at time 0. For clarity, we give a number to each block in the order they were created, although in practice nodes only use the hash of the blocks to compute the blockclique and do not rely upon their timestamps. We also show the parent relation in the graph with a gray link between a block and each of its parents. The links from the last created block to its parents are plotted in blue. A video showing this graph for one simulated hour is available at https://youtu.be/DhyLzxoWJM4.

3.1 Blockclique Architecture

To allow a higher transaction throughput, nodes must be able to produce compatible blocks in parallel. We define a fixed number of **threads** in which nodes can produce blocks. With a PoW random node selection mechanism, the nodes can choose in which thread they want to produce blocks (random by default), however a PoS mechanism can specify in which random thread a node is allowed to produce each block. To be sure that a transaction in a block is compatible with all transactions in a block discovered in another thread, we constrain threads so that each thread concerns a distinct subset of transactions, based on the input address of transactions. In other words, a given address should not be able to spend its coins in two different threads, as this could create a double spend. This process is called **transaction sharding**. For instance, if there are 32 threads, then the first 5 bits of an address define the thread in which the address can use its coins as input of a transaction. However, the output of a transaction can be any address, regardless of the thread corresponding to the input address. Also, at a high transaction throughput, the history of transactions grows fast, so the nodes should be able to forget part of the oldest blocks to save space. The nodes thus need to store the **balance** of each address, so that they can verify any transaction by checking if the input address has enough coins, without the need to look at old transactions. The next sections formalize the multithreaded block graph structure and the blockclique consensus rule.

3.1.1 A Multithreaded Block DAG

We define T as the number of threads, and each thread is identified by a number τ, from thread $\tau = 0$ to thread $\tau = T - 1$. Each block is identified by its hash: b_h^τ denotes the block with hash h in thread τ. We define one genesis block per thread, that we denote b_0^τ. Each non-genesis block must reference one parent block in each thread (using parent block's hash), so we define the parent function $P(b_{h_2}^{\tau_2}, \tau_1) \to b_{h_1}^{\tau_1}$ that gives the parent $b_{h_1}^{\tau_1}$ in thread τ_1 of a non-genesis block $b_{h_2}^{\tau_2}$. The parent function generates a block graph G where an arrow from $b_{h_1}^{\tau_1}$ to $b_{h_2}^{\tau_2}$ means that $P(b_{h_2}^{\tau_2}, \tau_1) = b_{h_1}^{\tau_1}$. As each block references its parents' hash and the hashing

procedure of a block takes into account those hashes, it is cryptographically impossible to build a cycle in this graph, so that G is a directed acyclic graph (DAG) of parallel blocks.

A block DAG is a **multithreaded block DAG** if first, non-genesis blocks reference one block of each thread and second, it has the additional property that a block $b_{h_2}^{\tau_2}$ cannot have in its ancestry a block b_h^τ in a thread τ that is a descendant of $P(b_{h_2}^{\tau_2}, \tau)$. Then, we define the generation function g_τ that gives the generation of a block b_h^τ in thread τ as the length of the only path in G between genesis block b_0^τ and b_h^τ going only through blocks in thread τ. We also define the $\texttt{Path}(G, b_{h_1}^{\tau_1}, b_{h_2}^{\tau_2})$ predicate, being true if there is a directed path in G between $b_{h_1}^{\tau_1}$ and $b_{h_2}^{\tau_2}$ or if $b_{h_1}^{\tau_1} = b_{h_2}^{\tau_2}$. We define the average time between two final blocks in each thread as t_0^f, and we denote t_0^v the average time if we consider all valid blocks, including stale blocks, so that if we call s the proportion of stale blocks, we have $t_0^v = t_0^f(1-s)$. Fig. 1 shows an example of a multithreaded directed acyclic graph of blocks.

3.1.2 Blockclique Rule

When a node receives a block from its peers, it first checks that the block is valid (see Section 3.2.2). Then it decides, based on the blockclique rule, which valid blocks it will take into account to produce a new block. The intuition of the rule is the following: on the first hand, each thread of blocks behave like a standard blockchain in the sense that two blocks in a same thread can't have the same parent in this thread (thread incompatibility), and on the other hand, we add a constraint so that nodes producing a block in a given thread take into account the blocks found in other threads and do not act as if their thread was independent (grandpa incompatibility).

We define the **thread incompatibility graph** G_{TI} as the graph with one node per valid block, and an undirected edge between two blocks $b_{h_1}^{\tau_1}$ and $b_{h_2}^{\tau_2}$ only if those two blocks are non-genesis blocks in the same thread and have the same parent in their thread:

$$G_{TI}(b_{h_1}^{\tau_1}, b_{h_2}^{\tau_2}) := \Big[\big[g_{\tau_1}(b_{h_1}^{\tau_1}) \geq 1\big] \texttt{ and } \big[g_{\tau_2}(b_{h_2}^{\tau_2}) \geq 1\big] \texttt{ and } \big[\tau_1 = \tau_2\big] \texttt{ and } \big[P(b_{h_1}^{\tau_1}, \tau_1) = P(b_{h_2}^{\tau_2}, \tau_2)\big]\Big] \quad (1)$$

Then we define the **grandpa incompatibility graph** G_{GPI} as the graph with one node per valid block, and an undirected edge between two blocks $b_{h_1}^{\tau_1}$ and $b_{h_2}^{\tau_2}$ if the parent of block $b_{h_2}^{\tau_2}$ in thread τ_1 is not the parent of $b_{h_1}^{\tau_1}$ or one of its descendants in τ_1 and the parent of block $b_{h_1}^{\tau_1}$ in thread τ_2 is not the parent of $b_{h_2}^{\tau_2}$ or one of its descendants in τ_2:

$$\begin{aligned} G_{GPI}(b_{h_1}^{\tau_1}, b_{h_2}^{\tau_2}) := \\ &\big[g_{\tau_1}(b_{h_1}^{\tau_1}) \geq 1\big] \texttt{ and } \big[g_{\tau_2}(b_{h_2}^{\tau_2}) \geq 1\big] \\ &\texttt{and } \big[\texttt{not Path}(G, P(b_{h_1}^{\tau_1}, \tau_1), P(b_{h_2}^{\tau_2}, \tau_1))\big] \\ &\texttt{and } \big[\texttt{not Path}(G, P(b_{h_2}^{\tau_2}, \tau_2), P(b_{h_1}^{\tau_1}, \tau_2))\big] \end{aligned} \quad (2)$$

The grandpa incompatibility is a topological way of saying that a block b_h^τ in thread τ should not be included if it does not take into account the blocks that were found in other threads before the time when $P(b_h^\tau, \tau)$ was found, but without checking block timestamps which can be inaccurate or manipulated. Fig. 2 shows an example of a grandpa incompatibility.

From those thread and grandpa incompatibility graphs G_{TI} and G_{GPI}, we can now define the **compatibility graph** G_C as the graph with one node per valid block, and an undirected edge between two blocks $b_{h_1}^{\tau_1}$ and $b_{h_2}^{\tau_2}$ if those two blocks are not thread or grandpa incompatible and if $b_{h_1}^{\tau_1}$ is compatible with the parents of $b_{h_2}^{\tau_2}$ and $b_{h_2}^{\tau_2}$ is compatible with the parents of $b_{h_1}^{\tau_1}$:

$$\begin{aligned} G_C(b_{h_1}^{\tau_1}, b_{h_2}^{\tau_2}) := &\big[\texttt{not } G_{TI}(b_{h_1}^{\tau_1}, b_{h_2}^{\tau_2})\big] \\ &\texttt{and } \big[\texttt{not } G_{GPI}(b_{h_1}^{\tau_1}, b_{h_2}^{\tau_2})\big] \\ &\texttt{and } \Big[\big[g_{\tau_2}(b_{h_2}^{\tau_2}) = 0\big] \texttt{ or } \\ &\big[G_C(b_{h_1}^{\tau_1}, P(b_{h_2}^{\tau_2}, \tau)) \texttt{ for all } \tau\big]\Big] \\ &\texttt{and } \Big[\big[g_{\tau_1}(b_{h_1}^{\tau_1}) = 0\big] \texttt{ or } \\ &\big[G_C(P(b_{h_1}^{\tau_1}, \tau), b_{h_2}^{\tau_2}) \texttt{ for all } \tau\big]\Big] \end{aligned} \quad (3)$$

The definition of G_C is recursive but G_C will be built incrementally following a topological order of G, by processing a block as soon as all its parents have been received and processed.

Let $\texttt{cliques}(G_C)$ be the set of maximal cliques of compatible blocks: the set of subsets C of blocks of G_C such that every two distinct blocks of C are adjacent in G_C and the addition of any block removes this property. We also define the work sum of a clique, $\texttt{worksum}(C)$ as the sum of the inverse of the targets of each block in C, and the hash sum of a clique, $\texttt{hashsum}(\mathrm{C})$ as the sum of the hash of each block in C. Now, the **blockclique rule** states that the best compatible block clique, the **blockclique**, is the clique of compatible blocks of maximum work sum, and if tie, of minimum block hash sum:

$$\texttt{maxworkcliques}(G_C) := \underset{C \in \texttt{cliques}(G_C)}{\arg\max} \Big[\texttt{worksum}(C)\Big] \quad (4)$$

$$\texttt{blockclique}(G) := \underset{C \in \texttt{maxworkcliques}(G_C)}{\arg\min} \Big[\texttt{hashsum}(C)\Big] \quad (5)$$

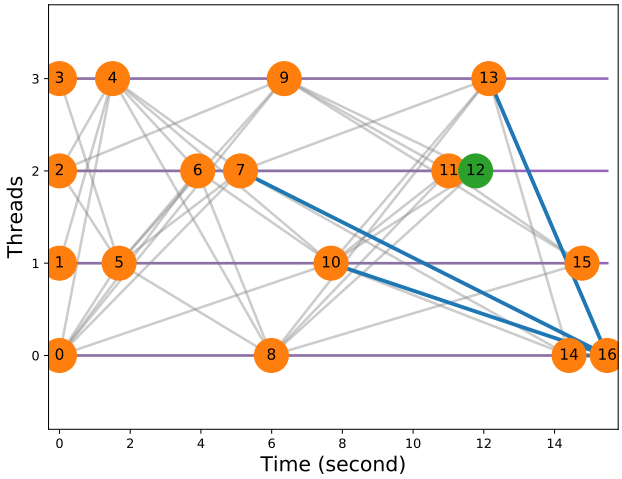


Figure 2: Example of a grandpa incompatibility in a block graph with 4 threads. In this example, the grandparent of block 12 is block 7, the grandparent of block 16 is block 8, the parent of block 12 in thread 0 is block 8 and the parent of block 16 in thread 2 is block 7. Equation 2 thus tells that there is a grandpa incompatibility between block 12 and block 16, so there is an edge in G_{GPI} between blocks 12 and 16. This happened because the node that produced block 16 did not receive blocks 11 and 12 before creating it.

3.1.3 Incremental Cliques and Finality

The problem of finding the maximum cliques of a graph is NP-hard, so in practice we can't quickly find the blockclique of the whole graph G once it has more than a few hundred blocks. We thus define an **incremental rule** for recomputing the cliques with only the most recent blocks. In practice, we keep in memory an incremental subgraph G_{INCR} of G by adding the new valid blocks and removing blocks when they can be considered stale or final. To do so, we remove the blocks that are only in cliques of G_C that have a size smaller or equal to the size of the blockclique minus a **finality parameter** F, which are considered stale blocks, and we remove the blocks that are common to all maximal cliques of G_C and have more than F descendants in at least one clique, which are then considered final blocks. We will also use the incremental version of G_{TI}, G_{GPI} and G_C by keeping the same nodes as in G_{INCR}.

This finality parameter F represents the lag with respect to the size of the blockclique from which an alternative clique is removed. With this definition, if an alternative clique has more than F blocks less than the blockclique, it still has a chance to survive if a majority of the nodes coordinate to switch to that clique. Consequently, the finality parameter F must be chosen high enough so that the probability for an alternative clique behind the blockclique by $F-1$ blocks to overtake the blockclique is very low even if an attacker has a non-negligible part of the mining power (PoW) or stake (PoS). On the other hand, if F is too large, then G_C has a lot of blocks and nodes need a long time to recompute the cliques each time they receive a block. Therefore F must be carefully chosen as a **tradeoff** between security and efficiency. See Sec. 3.3.2 for more details on this question. In the example of Fig. 3, we use $F = 64$ blocks, so the incremental block graph G_{INCR} has on average a bit more than 64 blocks, depending on the current stale rate.

When a new node wants to join the network and **synchronize** with other peers, it first downloads all the blocks from the peers. It cannot compute the maximal cliques of the whole block graph, so it directly computes the tip block (block without children) that has the maximum work in its ancestry. Then, the blockclique and some alternative cliques can be found by computing the cliques of only the most recent blocks: for instance the last 5 blocks in each thread in the ancestry of the tip block of maximum work, together with all their descendants. Starting with this blockclique and those alternative cliques, the node can use the incremental algorithm described above from now on. If by any chance an alternative clique rooted in an older fork becomes the blockclique according to the other peers, the node will observe a tip block with maximum work in its ancestry that is not included in its blockclique, in which case it can restart this synchronization procedure: recompute the tip block with maximum work and the blockclique of the most recent blocks with respect to this tip block.

3.2 Example Implementation of Transactions and Blocks with PoW

We give here a detailed implementation of the minimum data that must be stored in transactions and blocks to be able to validate transactions and apply the consensus rule, with a Proof-of-Work node selection mechanism.

3.2.1 Transactions

A transaction contains information about its version, the public key of its sender and its signature so that other nodes can verify the transaction, a fee value, an amount of coins transferred, a destination address, and some optional data. Table 1 shows the different fields of a transaction. The minimum size of a transaction is $1,016$ bits (127 B) if there is no data provided, and the maximum size is $1,048$ bits. The `VALUE` field of a transaction specifies the amount of tokens that must be sent to the receiving address, with a minimum amount of 10^{-8} tokens, and a maximum amount of $(2^{48}-1) * 10^{-8} = 2,814,749.76710656$ tokens. The transaction fee is chosen by the sender of the transaction, like in blockchains such as Bitcoin or Ethereum, so that when the network is under high load, transactions with higher fees should be processed first in the interest of miners.

The first transaction of the transaction list is called the **coinbase** transaction, and is used by the miner to claim the block reward. It has no `FROM_PK`, `FEE`,

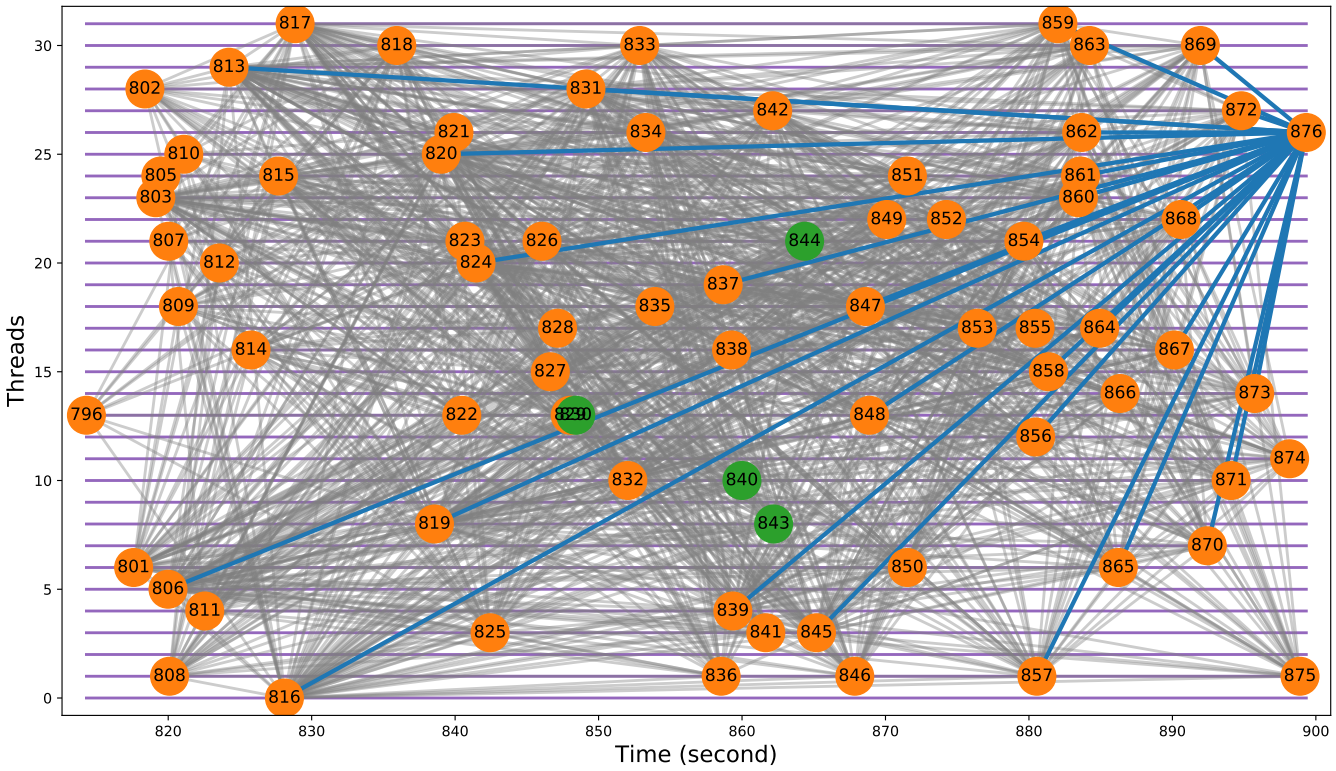


Figure 3: Incremental directed acyclic block graph G_{INCR} with a finality parameter $F = 64$ blocks. This simulation is the continuation of Fig. 1, about 15 minutes later. Blocks are removed from G_{INCR} as final blocks when they are common to all maximal cliques of the compatibility graph G_C and have more than F descendants in at least one clique, or as stale blocks when they are only in cliques of G_C that have a size smaller or equal to the size of the blockclique minus F. Once the genesis blocks are removed from G_{INCR}, this graph has on average a bit more than 64 blocks (with $F = 64$), depending on stale rate. In this example, there has been no block for a while in thread 2 and we can see that the parent in thread 2 of the last block 876 (block 792) is not in G_{INCR} anymore, because it was removed from G_{INCR} as a final block about 10 blocks earlier.

or `SIGNATURE` fields, and its destination address is the miner's address.

The input address of the transaction defines the particular thread in which this transaction can be included, however the output address can be in any thread. We show an example with only one input address and one output address for clarity, although the thread constraint allows transactions to have multiple inputs from addresses of the same thread, and multiple outputs in any threads.

3.2.2 Blocks

A block is composed of a block **header** plus a list of transactions. The minimum block size is 175 B, if there are no transactions, and the maximum block size is 1 MB, with a maximum of $7,872$ transactions of 127 B. The block hash function can be any secure hash algorithm. The hash is computed from the block header (from `VERSION` to `N_TRANSACTIONS` field), and has 256 bits. Table 2 shows the different fields of a block.

The first block of each thread is a **genesis** block, with a special status: it has no parents, so its `PREV_BLOCKS` field is filled with zeros. Its `NONCE` and `N_TRANSACTIONS` fields are also zeros (no transactions), its `TARGET` (see Sec. 3.2.3) is chosen based on an estimate of the mining power of the network at launch, and its `TIMESTAMP` field is the time of the launch of the network.

A block is considered **valid** if it follows several rules. Its size must be under the maximum block size (1 MB). The target field must be computed based on the timestamps of the previous blocks (see Sec. 3.2.3). The hash of the block must be under the target. The amount of the coinbase transaction must be the sum of the block reward (see Sec. 3.2.4) plus the fees spent in this block. The `PREV_BLOCKS` must comply with the multithreaded block DAG properties (see Sec. 3.1.1). The transactions in the block must all be valid, meaning that the input address of each transaction must have a balance, after applying the parent blocks of this block, larger than the total amount of coins spent by this address in this block. Finally, the root of the Merkle tree of the transactions in this block must be valid.

3.2.3 Mining Target

To mine a valid block, a miner has to find a nonce (number used once) that makes the 256 bits hash of the

TRANSACTION fields	Size (bit)	Description
VERSION	15	Version number
FROM_PK	256	Public key of the sender
TO_PKH	160	Public key hash (address) of the destination
VALUE	48	Amount of tokens transferred
IS_DATA	1	Tells if data is provided
(DATA)	(32)	Optional data
FEE	24	Fee paid to the miner (x10^{-8})
SIGNATURE	512	ECDSA-256 signature of the transaction by the private key of the sender

Table 1: Components of a TRANSACTION.

BLOCK fields	Size (bit)	Description
VERSION	14	Version number
THREAD	5	Thread of the block
PREV_BLOCKS	1,024	Reference to one block in each thread (e.g. last 32 bits of block hash x32 threads)
MERKLE_ROOT	256	Root of Merkle tree of transactions in block
TIMESTAMP	32	Unix timestamp recording when this block was created
TARGET	32	Hash target for this block
NONCE	24	Integer used when mining to change block hash until a hash below the target is found
N_TRANSACTIONS	13	Number of transactions in this block
TRANSACTIONS	$\leq$ 7,998,600	List of transactions (see Table 1)

Table 2: Components of a BLOCK.

block be below a **target** threshold. The target threshold can be encoded as a floating-point number that can reach 2^{256} with 32 bits, for instance with a 10 bits exponent and a 22 bits significand. To keep an average time t_0^f between two final blocks in each thread, the target for the current block being mined is updated at each block based on the number of blocks mined recently in all threads. The target is computed based on the timestamps of the last blocks in the time window in each thread including blocks until the parent of the current mined block in its thread, and potentially by removing 20% of the outliers so that inaccurate or fake timestamps have less incidence.

3.2.4 Block Reward

The total token supply can be bounded or infinite (with a long tail). We give here an example of a token supply bounded by $20,000,000$ tokens. The block reward can be defined in each thread with a curve smoothly decreasing at each new block in the thread. For example, it can be implemented as an exponentially decreasing curve that halves every 10 years, so that $10,000,000$ tokens are created during the first 10 years, $5,000,000$ tokens are created from year 10 to year 20, etc. The reward of a genesis block is $r_0 = 0$, the reward of the first block after genesis is $r_1 = 0.0438741$, and then the reward for the n-th block in a thread is $r_n = r_1 \times q^{n-1}$ with $q = 0.99999993$, and truncated at 10^{-8}. The estimated total supply $S(t)$ after time t (in seconds) is thus given by the sum of a geometric series in Equation 6.

$$S(t) = T \cdot r_1 \cdot \frac{1 - q^{t/t_0^f}}{1 - q} \tag{6}$$

The reward of a miner for creating a block is composed of both the block reward plus the sum of the fees of all the transactions included in this block.

3.3 Security

In this section, we study the different security properties of the blockclique architecture with respect to standard single-thread blockchains. The blockclique architecture needs the property that agents should prefer to follow the defined protocol for their own benefice, and that even if a large proportion of agents coordinate to attack the network (e.g. 30%), the network resists, the attackers get less rewards and honest agents do not.

First, it should be noted that in the blockclique architecture there are much more blocks created per second than in single-thread blockchains. There are T/t_0^f blocks becoming final per second on average, which is

about one block per second with $T = 32$ threads and $t_0^f = 32$s. In this example that would be 600 times more than in the Bitcoin network (one block every 10 minutes on average). In the Bitcoin network, a miner with say a few thousands dollars mining setup would need a few decades on average to discover one block, so miners have a great incentive to pool together to share the obtained rewards and reduce their reward variance. Pooling, however, reduces the decentralization of the network as only one node decides which transactions to mine and sends the block header to all the members of the pool. However, if there is one smaller reward distributed at each second on average, then the same miner would get more than one reward per month. Therefore, with the blockclique and with a PoW random node selection, there is much less incentive for honest miners to pool, so attackers can't use the mining power of honest miners and decentralization is protected.

In the following we first study the problem of double-spending, or why agents cannot spend the same coins multiple times. Then we explain how to choose the finality parameter for a good security-efficiency tradeoff to mitigate replay attacks and we discuss transaction finality. Also, with the blockclique architecture with a PoW random node selection, the miners can choose in which thread they mine blocks and the mining power is distributed across multiple threads, so it's easier for attackers to get more than 51% of the mining power in one thread and unbalance the network. We thus explain why this imbalance attack do not impair the security or liveliness of the network. However, a PoS random node selection can also specify in which thread a particular stake is allowed to produce a particular block, such that manipulating the threads of the produced block to unbalance the network is unfeasible. Finally, we show that the selfish mining attack of PoW blockchains doesn't work against the blockclique.

3.3.1 Double-Spending

In single-thread blockchains, nodes ensure that coins are not spent several times (or duplicated) by checking that the transactions in a block spend coins that have never been spent in previous blocks, and that are not spent in another transaction of the same block. Then attackers can still spend their coins in different incompatible blocks, for instance in two fork blocks having the same parent, but in the end, only one of those incompatible transactions will be included in the best chain.

With the blockclique architecture, nodes check that an amount sent from address A in a given block is an amount available to address A at this block, in other words that address A has received in the ancestry of this block more coins than it has sent and that are sent in this transaction. They also check that this transaction has not been processed before (in the ancestry of the block) and that other transactions in the same block do not spend more than the amount that remain after this transaction. The difference with single-thread blockchains is that here, the block containing the transaction can be compatible with other blocks in other threads, that may not have been discovered or observed yet. However, transaction sharding ensures that a transaction with a given input address is processed only in the thread corresponding to that address. The thread corresponding to an address is for instance determined by the $log_2(T)$ first bits of the input address. A node producing a block in a given thread is thus guaranteed that the amount of coins available to this address cannot be spent in parallel in a compatible block of another thread, therefore double-spending is avoided by the structure of the blockclique.

3.3.2 Replay attack

Assuming an honest majority, most of the mining/staking power is dedicated to the blockclique, such that an alternative clique behind the blockclique by a few blocks has a high probability to become more than F blocks late at some point in which case its blocks that are not in any other clique with less than F blocks late with respect to the blockclique will be considered stale.

However, given that the network latency (the time needed to broadcast a block to most of the peers, which we call $t_{1/2}$) can be of a few seconds (see [19] for an analysis of information propagation in the Bitcoin network) and that blocks appear for instance every second, then the different nodes of the network receive blocks in a different order. Consequently, it is possible that one node decides that a given block is stale because it is only in cliques that are F blocks or more behind the blockclique, while another node, having received one more descendant of this block would not consider it stale yet. If by chance, an alternative clique containing the block that is considered stale by some but not all of the nodes gets extended and eventually catch up with the blockclique, then the nodes that considered the block as stale will reject the blocks of the new blockclique and the network will fork. In order to avoid those forks, $\frac{F \times t_0^f}{T}$ must be high with respect to the network latency $t_{1/2}$: we will conservatively choose here the constraint $\frac{F \times t_0^f}{T} \geq 64$s.

Also, if a group of attackers with a non-negligible share of the total mining/staking power coordinates to switch to an alternative clique being $F - 1$ blocks late with respect to the current blockclique from the point of view of some nodes, and F blocks late from the point of view of other nodes, they can attack the network by trying to extend this alternative clique until it overtakes the current blockclique in order to create a fork in the network. This attack is called the replay attack. The finality parameter F must therefore be high enough so that this attack has a very low probability to succeed, even if the group of attackers has a large share (say 30%) of the total power. For instance, with $F = 64$ blocks, the probability for an alternative clique

behind the blockclique by 63 blocks to overtake the blockclique if an attacker with 30%, 40%, 50% or 60% of the power decides to switch to that clique is respectively about 10^{-24}, 10^{-12}, 1.6% and 33% if we consider that there are no stale blocks (see Appendix A). If we consider a stale rate of 25%, those probabilities become about 10^{-16}, 10^{-4}, 25% and 50%. A finality parameter greater than or equal to 64 is thus high enough to prevent this type of attack even when the stale rate is high, for a group of attackers of less than 30% of the total power. Note that this kind of attack is very costly as each time it fails, all the blocks created by the attackers become stale and are not rewarded.

Putting the security constraints together, we recommend to keep F high enough so that both $F \geq 64$ and $\frac{F \times t_0^f}{T} \geq 64$s. However, if F is too large, then the incremental compatibility graph G_C has too much blocks and nodes may need a non-negligible time to recompute the cliques. We thus consider here that a good security-efficiency tradeoff is $F = 64$ blocks with for instance $T = 32$ blocks and $t_0^f = 32$s.

3.3.3 Transaction Finality

Now, with those parameters, how much time do we need to wait to be sure (with a given probability) that a transaction in a given block will always be considered final ? In single-thread blockchains, if one wants to be sure that the transaction will be included in the best chain one can wait that several blocks have been discovered on the branch containing the transaction. The higher the required certainty, the higher the required number of blocks appearing in the good branch. For instance, waiting for a confirmation of 6 blocks (one hour on average) is often recommended in the Bitcoin network.

With the blockclique architecture, to be sure that a transaction will stay included in the blockclique one can wait that the block containing the transaction has been removed from the incremental graph as a final block. This happens when the block belongs to all maximal cliques of the compatibility graph and has more than F descendants in at least one clique. Indeed, if an attacker then tries to broadcast a new block that is incompatible with the block containing the transaction, this new block would only be in a clique which is more than F blocks late with respect to at least one clique and so with respect to the blockclique, and would directly be considered stale by honest nodes.

Note that for low-cost transactions, one (say a baker selling a baguette) can wait less than F blocks to consider the transaction confirmed with high probability. Indeed, by checking that the transaction is included in a block included in the current blockclique and by waiting that all the alternative cliques that do not contain the transaction become e.g. 16 blocks late with respect to the blockclique, instead of 64 blocks, the expected total time (from broadcasting) for this transaction will be about 1 minute instead of 126 seconds in a network with $T = 32$ blocks, $t_0^v = 32$s, $F = 64$ blocks and $U = 32$ Mbps (see Sec. 4.2), while the probability for the blockclique to be overtaken by another clique not containing the transaction would still be very low (see Appendix A with e.g. $F = 16$ blocks).

3.3.4 Imbalance Attack

With the blockclique architecture with a Proof-of-Work random node selection mechanism, the mining power is distributed across multiple threads, and miners are free to choose one or several particular threads in which they will mine blocks, however, we define a default behavior. The default behavior for small miners is to mine in a single thread at a time and regularly move to a random thread. Miners with more resources can mine in several random threads at the same time and regularly change threads or even mine in all threads if their transaction processing power and network capacity is high enough to allow it.

Then, if a single attacker with a large proportion of the total mining power decides to stay in a single thread, or if a group of attackers coordinate to mine in a single specific thread instead of being randomly spread over several threads, this attack will unbalance the mining power across threads. We call this attack the imbalance attack. Consequently, there will be a higher block frequency in the attack's thread than in other threads, because the block target is a variable computed globally and is not adapted to the block frequency in each thread.

It is much easier for a group of attackers to have 51% of the mining power of a single thread than 51% of the overall mining power. For instance, an attacker with 20% of the total mining power in a blockclique architecture with $T = 32$ threads can have 89% of the mining power of one particular thread, if we suppose that the other miners are evenly spread across all the threads. With the imbalance attack, the attackers can thus have a large majority of the mining power in one specific thread. In this configuration they can 1) mine normal blocks, 2) mine void blocks, or 3) mine blocks excluding specific transactions.

In the first case, we simulate the consequences of this attack depending on the total mining power proportion of an attacker, from 0.5% to 20%, and depending on the level of transaction load: with a non-saturated level of 800 transactions per second and with a saturated level of 8000 transactions per second (see Sec. 4.2.4). Fig. 7 (a) shows the rate of final blocks of the attacker and of other miners under the imbalance attack with a non-saturated transaction load, for $T = 32$ threads and $t_0^v = 32$s. Fig. 7 (b) shows the rate of final transactions, while Fig. 7 (c) and (d) show the rate of final blocks and transactions with a saturated transaction load. Those simulations show that when the mining power proportion of the attacker goes from

0.5% to 20%, the rate of final blocks of the attacker per percentage of mining power slightly increases, while its rate of final transactions largely decreases, both with a non-saturated or saturated transaction load, whereas the rate of final blocks of honest miners that regularly change to a random thread slightly increase and their rate of final transactions also increases. The cost of the attack to the attacker thus depends on the relative value of transaction fees and block rewards, however the higher the attacker mining power, the higher the number of final blocks and transactions and thus the rewards of honest miners.

In the second case, the attackers decide to mine blocks with no transactions. In this case, the honest nodes in the attack's thread normally mine transactions, even if they have a minority of the mining power in the thread. Those void blocks could increase a bit the rate of stale blocks in this thread as the block frequency gets higher, however, as they are very small (175 B), they are verified and broadcasted much faster than full blocks so this attack would have less impact than the previous one.

In the third case, the attackers could also try to prevent some specific transactions from being processed. In that case, the honest nodes in the same thread would process them with almost the same block frequency (the stale rate can be higher in this thread) as in other threads, so the liveliness of the network is not impaired.

3.3.5 Selfish Mining Attack

Finally, the attackers could try to implement a selfish mining attack [20]. In a selfish mining attack on a single-thread blockchain with a PoW random selection mechanism, a group of attackers with a large share of the total mining power (say 25%), can keep the blocks that they mined private, continue mining on top of those blocks, and broadcast them one by one when the other nodes of the network find blocks and are about to catch up. If certain conditions are met, the honest nodes of the network spend a non-negligible time mining blocks that have more chance than usual to become stale, so that their mining efficiency is reduced and the one of the attackers is mechanically increased.

With the blockclique, selfish attackers could even have a majority of the mining power in a specific thread. However, the grandpa incompatibility rule (see Sec. 3.1.2) specifies that a block should take into account (as parents) in other threads the blocks that were discovered at least up to when its parent (in its thread) was discovered. Thus, honest miners of all threads take into account blocks of all threads as soon as they observe and verify them, and as the blockclique rule specifies that the blockclique is the clique of maximum work sum (and if tie, of minimum hash sum), honest miners are "voting" for the best clique and thus for blocks in all threads, including the one where attackers have a majority. Selfish miners are thus playing against the mining power of the whole network of miners so the fact that they can have a large majority of the mining power in a specific thread do not facilitate the selfish mining attack compared to the attack on a single-thread blockchain.

Also, selfish attackers mining a block b_h^τ on top of a non-broadcasted block $P(b_h^\tau, \tau)$ have a higher than usual probability that their block b_h^τ becomes stale. Indeed, the new blocks of other threads will not take into account the non-broadcasted block $P(b_h^\tau, \tau)$ and will instead take into account its parent, block $P(P(b_h^\tau, \tau), \tau)$ which is the grandparent of b_h^τ. To avoid grandpa incompatibilities, the block b_h^τ should thus take into account at least the parent of all those blocks referencing its grandparent, and to do so, the block b_h^τ cannot be mined ahead of its time. Also, the selfish mining attack works when the honest miners mine on top of the first block they received in the case where they received two incompatible blocks, but do not work when miners randomly sample one of the incompatible blocks. However with the blockclique consensus rule, the honest miners will not take into account the first block they received but the one with the lowest target, and if tie, with the lowest hash, which is one of them in particular, defined by the random hash of the two blocks, assuming that two thread incompatible blocks that have the same parent in their thread have the same target.

3.4 Resources Analysis

In this section we analyze the need for computing, network and storage resources depending on the blockclique parameters and show that the high transaction throughput can be reached with fair individual node resources, and that with the blockclique architecture those three resources are more balanced than with the blockchain architecture.

3.4.1 Storage

The nodes contributing to the network need to store two main types of data: the blocks of transactions and the balance of addresses. At the average rate of one block per second, there are on average $31,557,600$ blocks created per year. Block headers have a size of 175 B which makes 5.5 GB of headers per year, and the maximum size of transactions is $7,998,600$ bits per block, which makes a maximum of 31.6 TB of transactions per year. Any node can store all block headers, but only some of the nodes will be able to store all block transactions. Therefore, nodes may only store the most recent blocks, and depending on their storage capacity, a random subset of the oldest blocks. For instance, a node with a standard storage capacity of 2 TB can store all block headers and 1% of 5 years of block transactions.

The addresses holding a non-zero balance are stored in an address-balance dictionary. The size of a balance can be 48 bits, which means that each address can hold

a maximum of $(2^{48}-1)*10^{-8} \approx 2,814,749$ tokens. The size of an address is 160 bits, so each address-balance weighs 208 bits. We can thus store about 10 billions non-empty addresses with 260 GB, which is a reasonable number of addresses considering that each person or merchant can use the same address multiple times. Indeed, each output of a transaction has an optional 32 bits data field, which means that the receiver of a transaction (say a merchant selling a product) has a way to identify the sender (the client) by giving him a 32 bits identifier to put in the data field. The merchant can then change its address only once in a while for security or anonymity purposes. Also, to avoid spamming by creating a lot of new addresses with very small balances, we can add a constraint on transactions which states that a transaction cannot leave an input or output address with a non-zero balance lower than the fee of the transaction.

When synchronizing from scratch, a new node either needs to download all the blocks, which will take a significant time after a few years of full blocks, or trust a third-party to download directly from it the address-balance dictionary up to some point, and then only download the blocks from this point from the peer-to-peer network.

3.4.2 Computation

Each node has to perform some operations fast enough to keep up with the flow of transactions and blocks. For each block, the nodes must check that the hash of the block is below its target, which takes about 30 ms. Also, the ECDSA signature of each transaction must be verified, which takes about 100 us per transaction, and checking the balance of senders of transactions requires some database accesses (read: 1 us, write: 10 us), so a cpu with 4 cores can check $8,000$ transactions in about 250 ms. Finally, the blockclique must be updated for each new valid block, which takes on average 10 ms with $F = 64$. An average cpu with several cores can thus easily verify an average flow of 1 block of $8,000$ transactions per second.

3.4.3 Network Connectivity

Nodes are connected through a peer-to-peer network to share blocks and transactions with other nodes. Each node needs to download the blocks that are found by other nodes, with an average stream of one block per second (8 Mbps) in our example, plus new unconfirmed transactions in the thread where the node is mining, but those are small compared to the size of the blocks. Nodes also contribute to the network by sending block headers and block transactions to other peers that request them. If on average each node sends each block to one other peer, then nodes need at least 8 Mbps of upload bandwidth. However, for the network to synchronize quickly enough to keep the rate of stale blocks reasonable, the higher the average upload bandwidth of nodes the better. In the Bitcoin and Ethereum networks, the median upload bandwidth of nodes is estimated to 56.1 Mbps and 29.4 Mbps respectively [21], and the median latency between nodes to 109 ms and 152 ms respectively. In our simulations we consider different conditions for the average upload bandwidth of miners: from medium speed (average of 32 Mbps) to fiber-level speed (average of 128 Mbps), and for the average latency: from an average of 50 ms to an average of 150 ms.

4 Simulations

In this section, we describe the methods of our simulations with the blockclique architecture and the results of several experiments with a Proof-of-Work random node selection mechanism.

4.1 Methods

In order to evaluate the blockclique architecture, we simulate the interaction of nodes in a peer-to-peer network, that discover blocks, ask and send blocks to peers and compute the blockclique. The code for these simulations is written in Python and is open-source[1].

4.1.1 Peer-to-peer Network

In these simulations, the peer-to-peer network is generated as a directed graph of N nodes with random connections between peers. Each node has a particular upload bandwidth u for sending blocks, randomly sampled at the beginning of the simulation between $\frac{1}{2}U$ and $\frac{3}{2}U$ where U is the average upload bandwidth of all nodes, and sends blocks one by one sequentially at the maximum speed of its upload bandwidth, and with a random latency depending on the destination node. The latency between two nodes is sampled at the beginning of the experiment between 0 ms and $2L$ where L is the average latency between two nodes of the network. Each node is connected to a number of random successors s that depends on its upload bandwidth u: $s = \lfloor u/8 \rfloor$ with u in Mbps. If by this random process one of the nodes has no predecessors, we also connect this node to its successors both ways, so that all nodes have at least one predecessor.

We implement the trick described in [19]: when a node discovers or receives a block, it first proposes the block to each of its connected peers before verifying the block, by sending a message (again through a connection with the given latency). The node then verifies the block and its transactions before forwarding the block to its successors. Those verifications are simulated in the sense that we only compute the time needed for the node to verify the block and the transactions. The

[1] https://github.com/sebastien-forestier/blockclique

block verification time is 50 ms, and the transaction verification time is 0.025 ms per transaction included in the block. Then, when a node receives the message saying that a block is available, if this node never asked for this block, it does now by sending back a message asking for the block (with again some latency). Finally, when a node receives a message from a successor asking for a block, it adds the request to its sending queue and it will send the block when its connection is available.

We also implement other small optimizations: when a node receives a block, it informs its successors only when its sending queue is almost cleared, so that those successors have a chance to ask the block to another node that is readier to send, and a node preferentially sends the blocks it created versus blocks of other nodes.

Each event in this network, such as the reception of a block or of a message is included in an event queue ordered by the timestamp of the events. The main simulation loop consists in processing the network events one by one, by adding new events into the queue when a new message or block is sent, and by updating the blockclique from the point of view of the node that receive a block.

4.1.2 Threads and Mining

Each node is assigned to a particular thread to mine new blocks, based on its identifier, so that exactly $\frac{N}{T}$ nodes are assigned to each thread, although in reality mining nodes would by default mine in a random thread and change thread randomly from time to time. We assume here that each node has the same constant mining power, and we do not simulate the actual mining algorithm nor the target hash to reach. We simulate mining by computing the time when each node will find a block with an exponential law of scale $\beta = \frac{t_0^v \times N}{T}$ so that one block is discovered on average every t_0^v seconds in each thread, although in reality we would specify t_0^f and adapt block targets to achieve it.

To save memory in this simulation, the blocks are stored once in a global dictionary, but each node processes those blocks only when received. Nodes also forget all the old blocks that are now useless to them: their block graph G is kept as incremental as possible although in reality nodes could store as much blocks as their storing space would allow.

To speed up the simulation, we separate the processing of blocks by the mining nodes into T parallel processes based on the thread number, that communicate by sending blocks and other information through Python pipes.

4.1.3 Blockclique Computation

When a node receives a block, it updates the blockclique based on the algorithm described in Section 3.1: it computes the clique of blocks of the compatibility graph G_C with the maximal worksum and if tie, the minimal hashsum. Based on this updated blockclique and the corresponding block references, it updates its mining process with a new block header and number of transactions. However, in those simulations we used a previous version of the grandpa incompatibility rule (Eq. 7), while the new version (Eq. 2) is simpler and also takes into account the cases where two blocks points to an ancestor older than the grandparent, which is also considered an incompatibility with the new version of the rule, although in practice those cases rarely happen:

$$
\begin{aligned}
&G_{GPI}(b_{h_1}^{\tau_1}, b_{h_2}^{\tau_2}) := \\
&\quad \Big[\big[g_{\tau_1}(b_{h_1}^{\tau_1}) \geq 1\big] \texttt{ and } \big[g_{\tau_2}(b_{h_2}^{\tau_2}) \geq 2\big] \\
&\qquad \texttt{and } \big[P(b_{h_1}^{\tau_1}, \tau_2) = P(P(b_{h_2}^{\tau_2}, \tau_2), \tau_2)\big] \\
&\qquad \texttt{and } \big[\texttt{not Path}(G, P(b_{h_1}^{\tau_1}, \tau_1), P(b_{h_2}^{\tau_2}, \tau_1))\big]\Big] \\
&\quad \texttt{or } \Big[\big[g_{\tau_1}(b_{h_1}^{\tau_1}) \geq 2\big] \texttt{ and } \big[g_{\tau_2}(b_{h_2}^{\tau_2}) \geq 1\big] \\
&\qquad \texttt{and } \big[P(b_{h_2}^{\tau_2}, \tau_1) = P(P(b_{h_1}^{\tau_1}, \tau_1), \tau_1)\big] \\
&\qquad \texttt{and } \big[\texttt{not Path}(G, P(b_{h_2}^{\tau_2}, \tau_2), P(b_{h_1}^{\tau_1}, \tau_2))\big]\Big]
\end{aligned}
\tag{7}
$$

4.1.4 Transactions and Blocks

We also simulate one transaction pool per thread, shared for each nodes of the same thread: when a node receives a block, it updates the blockclique and looks at the transaction pool of its thread for how much unconfirmed transactions are available based on its new block references. At this point we update the number of transactions in this pool based on a Poisson distribution of parameter $\lambda = \frac{t \times TL}{T}$ where t is the time elapsed since the last update of this pool, and TL is the transaction load: the average total number of transactions observed per second in all threads, which we set in the following experiments at $TL = 800$ or $TL = 8,000$ transactions per second.

The maximum size of a block BS depends on the number of threads T and the time t_0^v between two blocks in each thread, so that about $8,000$ transactions per second can be included in blocks: $BS = \frac{8,000,000 \times t_0^v}{T}$ bits. The size of a block header, BHS, is 376 bits plus the size of the `PREV_BLOCKS` field, so $BHS = 376 + T \times 32$. The maximum number of transactions per block is thus $TPB = \frac{BS - BHS}{TS}$, where the size of a transaction is assumed here to always be $TS = 1016$ bits. For instance, with $T = 32$ and $t_0^v = 32$ seconds, we simulate a block size of 1 MB, and the maximum number of transactions per block is $7,872$.

4.2 Results

We carry out four experiments to study the influence of different factors on the efficiency of the blockclique architecture. We vary the blockclique structure parameters (number of threads, time between two blocks), the network connectivity (bandwidth, latency), the network size (number of nodes), and we simulate the imbalance attack. We measure the average transaction throughput, the average block stale rate and the average confirmation time for the duration of the simulations. A video of the incremental block graph in one of these simulations ($T = 32$, $t_0^v = 32$) is available online[2].

4.2.1 Structure Parameters

In a first experiment, we vary the number of threads T between 16 and 64 threads, the time between blocks t_0^v between 16s and 64s, the block size between 2 Mb and 32 Mb as a function of T and t_0^v so that about $8,000$ transactions per second can be included in blocks, and the average upload bandwidth U between 32 Mbps and 64 Mbps. The finality parameter F is set to 64 blocks, and the transaction load to $TL = 8,000$ txps. We simulate here a network of $1,024$ nodes starting from T genesis blocks and until $3,600$ blocks are created in total. We run 10 trials of each condition by seeding the network random generation with the trial number so that 10 different random peer-to-peer networks are simulated. We also seed the numpy instances of the T Python processes with a combination of the trial and thread numbers.

The **average transaction throughput** is the number of transactions included in total in the blockclique after $3,600$ blocks created, divided by the time elapsed in seconds. Fig. 4 (left) shows the average and standard deviation over trials of the average transaction throughput, depending on the number of threads T, the time between blocks t_0^v, and the average upload bandwidth of nodes U (top: $U = 32$ and bottom: $U = 64$), with an average latency between nodes of $L = 100$ ms.

First, we can see that the average transaction throughput increases with the average upload bandwidth of nodes, for all combinations of T and t_0^v. For example, with $T = 32$ threads and $t_0^v = 32$s, the average throughput is $6,285$ txps and $6,997$ txps for respectively $U = 32$ and 64 Mbps. Also, with $U = 32$ Mbps, the average throughput increases with the number of threads from $T = 16$ to $T = 64$, whereas for $U = 64$ Mbps, the average throughput is similar with $T = 32$ and $T = 64$. Finally, the average transaction throughput increases with the time between two blocks t_0^v, for any combination of T and U.

The **average block stale rate** is the proportion of blocks that are not included in the blockclique, computed after the creation of $3,600$ blocks. Fig. 4 (middle) shows the average and standard deviation over trials of the average block stale rate in the same experiments, depending on the number of threads T, the time between blocks t_0^v, and the average upload bandwidth of nodes U, with $L = 100$ ms.

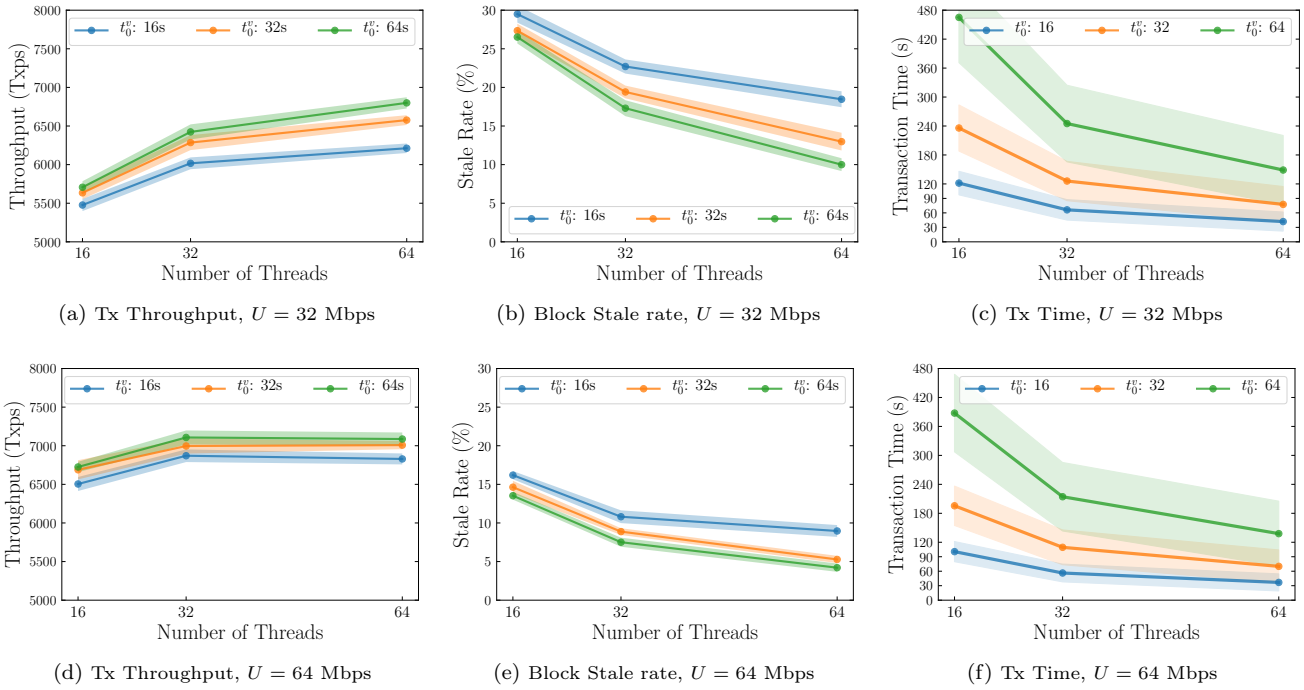
First, the stale rate decreases with the average upload bandwidth of nodes, for any combination of T and t_0^v. For example, with $T = 32$ threads and $t_0^v = 32$s, the average stale rate is 19.4%, and 8.9% for respectively $U = 32$ and 64 Mbps. The average stale rate also decreases with the time between two blocks t_0^v for any combination of T and U, and with the number of threads for any combination of t_0^v and U.

The average stale rate is negatively correlated with the average transaction throughput: the lower the stale rate, the higher the transaction throughput. However this is not a perfect correlation, as we can see for instance by comparing Fig. 4 (d) and (e), for example with $t_0^v = 64$s: the average transaction throughput with $T = 32$ and $T = 64$ blocks is respectively $7,107$ txps and $7,087$ txps, whereas the average stale rate is respectively 7.5% and 4.2%. This can be due to a different size distribution of the stale blocks depending on the number of threads T: the size distribution of stale blocks can be more skewed towards larger blocks with $T = 64$ than with $T = 32$, so that the transaction throughput can be higher with $T = 32$ threads while the block stale rate is also higher.

The **average total transaction time** is the time needed for a new transaction to be included in a block, plus the confirmation time: the time for this block to become a final block. We assume here that the blocks are not saturated so that there is no waiting queue for transactions. When a transaction is broadcasted to the miners in the thread of this transaction, we assume that miners take this transaction into account as soon as they receive a new compatible block, which takes on average t_0^f/T seconds, and create a block that will become final and that includes this transaction after an average time t_0^f. Then, the time needed for this block to become final, which we call the confirmation time, is measured for each final block and averaged. Fig. 4 (right) shows the average over trials of the average total transaction time in the same experiments, depending on the number of threads T, the time between blocks t_0^v, and the average upload bandwidth of nodes U, with $L = 100$ ms. We also plot with a shaded area the average over trials of the standard deviation of the total transaction time due to the variability of the time for a new transaction to be included in a final block (estimated) and the variability of the confirmation time of a block (measured).

First, the total transaction time decreases with the average upload bandwidth of nodes, for any combination of T and t_0^v. For instance, with $T = 32$ threads and $t_0^v = 32$s, the average total transaction time with $U = 32$ Mbps is 126s with a standard deviation of 41s, and with $U = 64$ Mbps is 110s with a standard devia-

[2] https://youtu.be/DhyLzxoWJM4



(a) Tx Throughput, $U = 32$ Mbps (b) Block Stale rate, $U = 32$ Mbps (c) Tx Time, $U = 32$ Mbps

(d) Tx Throughput, $U = 64$ Mbps (e) Block Stale rate, $U = 64$ Mbps (f) Tx Time, $U = 64$ Mbps

Figure 4: Average transaction throughput (left), block stale rate (middle) and total transaction time (right) depending on the number of threads T, the time between blocks t_0^v, and the average upload bandwidth of nodes U (top: $U = 32$ Mbps, bottom: $U = 64$ Mbps). We simulate here a network of $1,024$ nodes starting from T genesis blocks and until $3,600$ blocks are created in total. The average latency between nodes is $L = 100$ ms. The finality parameter is set to $F = 64$ blocks. We run 10 trials of each condition with different seeds and show the average and standard deviation of the three measures across trials.

tion of 36s. This is due to the fact that with a better network connectivity, the stale rate is lower and blocks become final faster because they get more than F descendants in at least one clique faster (see Sec. 3.1.3). Also, if the number of threads T increases, other things held constant, then the number of blocks discovered per second gets higher, and so the time needed for a block to become final gets lower, but the standard deviation do not decrease much because it is mainly due to the time needed for the transaction to be included in a block. Finally, the higher the time between blocks t_0^v, the higher the average total transaction time and its standard deviation, because the time needed for a transaction to be included in a block that will become final is on average $t_0^f/T + t_0^f$.

4.2.2 Bandwidth and Latency

In a second experiment, we study the variation of the average transaction throughput and block stale rate with respect to the average bandwidth and latency between two nodes in the peer-to-peer network. We vary the average latency L between 50 ms and 150 ms and the average upload bandwidth U between 32 and 128 Mbps, with the other parameters held constant: $T = 32$ threads, $t_0^v = 32$s, $N = 1,024$ nodes. With a given average latency L, the particular latency chosen to send blocks and messages from one node to another is randomly sampled between 0 and $2L$ at the beginning of the trial. We run 10 trials of each condition. Fig. 5 shows the average transaction throughput, block stale rate and transaction time depending on the average latency L and the average upload bandwidth U.

First, as before, the average transaction throughput increases and the average block stale rate decreases with the average upload bandwidth U, for any average latency L. Also, if the average latency increases from $L = 50$ ms to $L = 150$ ms, then the average transaction throughput slightly decreases with $U = 32$ and 64 Mbps and stays similar with $U = 128$ Mbps, and the average stale rate slightly increases for any U. For instance, with an average upload bandwidth $U = 32$ Mbps, the average throughput decreases from $6,361$ txps to $6,165$ txps, and the average stale rate increases from 17.7% to 20.6%. The average transaction time slightly increases with the average latency and decreases with the average upload bandwidth.

4.2.3 Network Size

In a third experiment, we study the variation of the average transaction throughput and block stale rate depending on the network size, for several average upload bandwidth U. Here we keep other parameters constant,

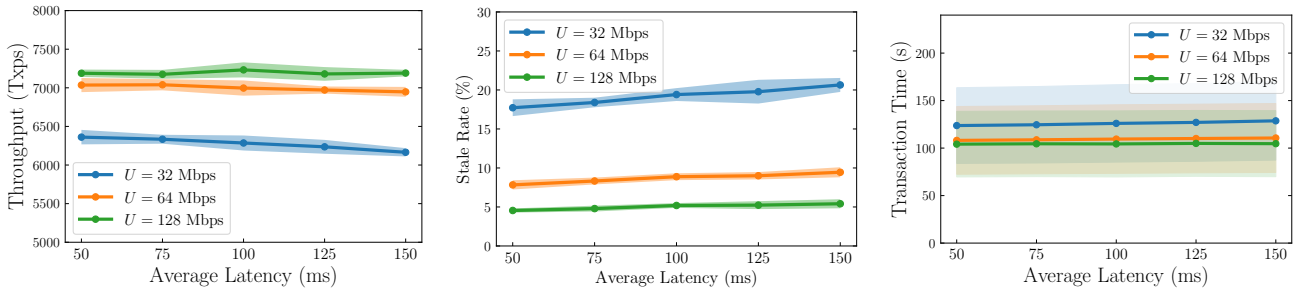


Figure 5: Average transaction throughput (left), block stale rate (middle), and transaction time (right) depending on the average bandwidth U and latency L between two nodes of the network, with $T = 32$ threads, $t_0^v = 32$s, and $N = 1,024$ nodes. We plot averages and standard deviations over 10 trials with different random seeds.

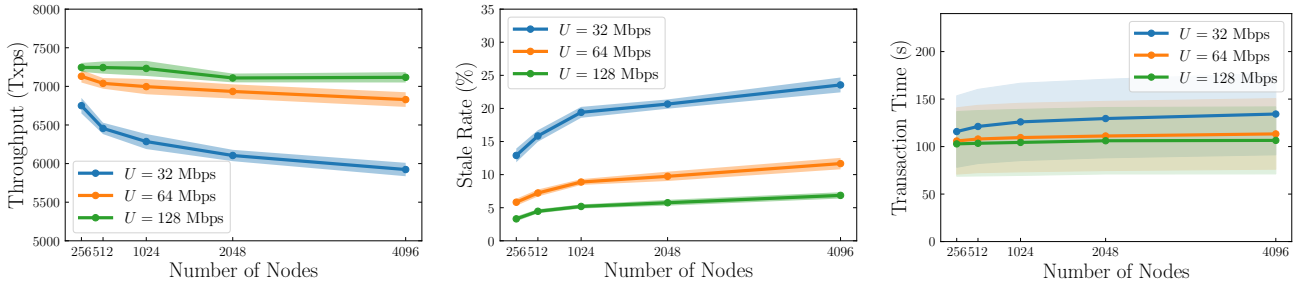


Figure 6: Average transaction throughput (left) and block stale rate (middle), and transaction time (right) depending on the number of nodes in the network and the average bandwidth U between two nodes of the network, with $T = 32$ threads, $t_0^v = 32$ seconds and $L = 100$ms. We plot averages and standard deviations over 10 trials with different random seeds.

with $T = 32$ threads, $t_0^v = 32$ seconds, $L = 100$ ms. We also run 10 trials of each condition. Fig. 6 shows the average transaction throughput, block stale rate and transaction time depending on the number of nodes in the network, from 256 nodes (8 nodes per thread), to $4,096$ nodes (128 per thread).

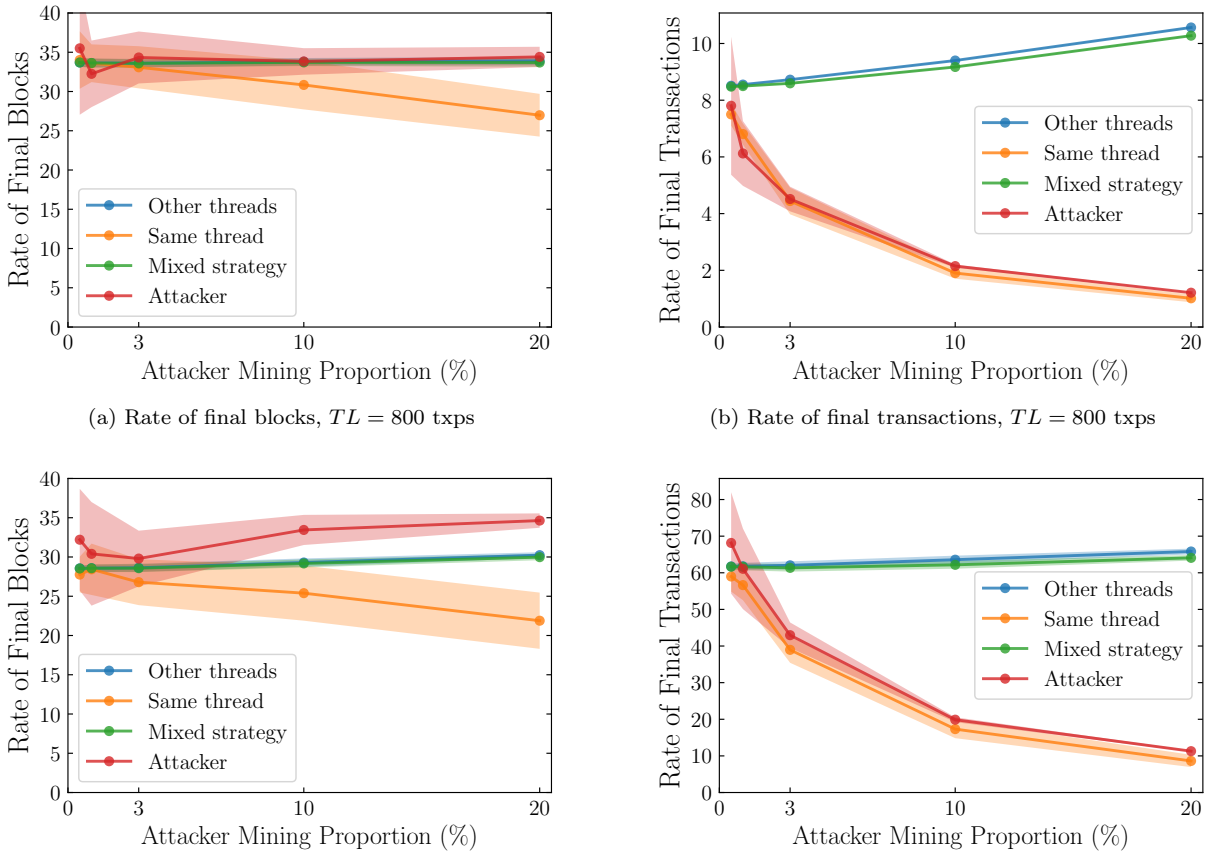
Here again, the average transaction throughput increases and the average stale rate decreases with the average upload bandwidth. For instance, with $1,024$ nodes, the average throughput with $U = 32$, 64 and 128 Mbps is respectively $6,285$ txps, $6,997$ txps and $7,232$ txps, and the average stale rate is respectively 19.4%, 8.9% and 5.2%. Then, the average transaction throughput decreases and the average stale rate increases with the number of nodes. For instance, with $U = 32$ Mbps and with a number of nodes of 256, 512, $1,024$, 2048, and 4096 the average throughput is respectively $6,750$ txps, $6,456$ txps, $6,285$ txps, $6,105$ txps, and $5,923$ txps, and the average stale rate is respectively 12.9%, 15.8%, 19.4%, 20.7% and 23.6%. The average transaction time increases with the number of nodes and decreases with the average upload bandwidth.

4.2.4 Imbalance Attack

In this experiment, we study the imbalance attack where one miner with a fair proportion of the total mining power decides to mine in only one thread instead of dividing its mining power into several threads at the same time or switching between threads from time to time, granting himself a large majority of the mining power in this particular thread.

We vary the mining power proportion P_A of the attacker with respect to the whole network from 0.5% to 20% across different simulations, while the rest of the mining power is divided between the other $1,023$ honest miners. The upload bandwidth of the attacker is set to 128 Mbps, and the average upload bandwidth of honest miners is 32 Mbps. The honest miners are balanced across threads and stay in their threads for the whole duration of the simulation. We also vary the blockclique structure parameters: the number of threads T from 16 to 64 threads and the time between two blocks t_0 from 16s to 64s when $t_0 > T$. Finally, we control the load of transactions TL that appear in the transaction pools at each second, with a non-saturated level $TL = 800$ txps and a saturated level $TL = 8,000$ txps. The average latency between two nodes of the network is $L = 100$ ms. We run 20 trials for each combination of P_A, T, t_0^v and TL, and each trial lasts until $3,600$ blocks are created.

We measure the rate of final blocks as the number of final blocks mined per hour and per percentage of the total mining power, and the rate of final transactions as the number of transactions included in final blocks per second and per percentage of the total mining power. We average this measure for three groups of miners: the attacker (1 miner), the honest miners in the same thread



(a) Rate of final blocks, $TL = 800$ txps

(b) Rate of final transactions, $TL = 800$ txps

(c) Rate of final blocks, $TL = 8000$ txps

(d) Rate of final transactions, $TL = 8000$ txps

Figure 7: Rate of final blocks of the different miners under the imbalance attack and a non-saturated transaction load $TL = 800$ txps, depending on the attacker's proportion of mining power. The rate of final blocks is the number of blocks mined per hour and per percentage of the total mining power. The different miners are the attacker, the miners that stay in the attacker's thread, the miners that stay in other threads, and miners that would have used a random mixed strategy. We plot the average and standard deviation over 20 runs with different random seeds. This figure shows the results for $T = 32$ threads and $t_0^v = 32$s. See Appendix B for results of other structure parameters T and t_0^v.

of the attacker (31 miners), and the honest miners in the other threads (992 miners).

We also compute the rate of final blocks and of final transactions of a miner that would have used a random mixed strategy: mining with the same power in each thread or mining in one thread but randomly changing thread with a enough frequency. Those rates are computed as a weighted average of the rate of the miners in the same thread as the attacker (with weight 1) and the rate of the miners in other threads (with weight $T-1$).

Fig. 7 (a) shows the average and standard deviation over 20 trials of the rate of final blocks mined per hour and per percentage of mining power for the attacker, the miners in the same thread as the attacker, the miners in other threads and the miners that would have used a random mixed strategy, depending on the attacker mining proportion P_A being 0.05%, 1%, 3%, 10% or 20%, with $T = 32$ threads and $t_0^v = 32$s, with a non-saturated level of transaction load $TL = 800$ txps. Fig. 7 (c) shows the rate of final blocks with a saturated level of transaction load $TL = 8,000$ txps. Fig. 7 (b) and (d) show the rate of final transactions with a non-saturated transaction load $TL = 800$ and with a saturated transaction load $TL = 8,000$ txps. The standard deviation is higher for miners of the same thread as the attacker because they are fewer and thus mine less blocks than miners of the other threads, and the standard deviation for the attacker is higher when P_A is low because it mines fewer blocks. See Appendix B for similar results with other structure parameters T and t_0^v.

First, for all tested combinations of T and t_0^v, the rate of final blocks of miners staying in other threads and of miners in the mixed strategy slightly increases or stays similar when the attacker's proportion of mining power P_A increases. For instance with $T = 32$ threads and $t_0^v = 32$s, Fig. 7 (a) shows that with P_A being 3%, 10% and 20%, the rate of final blocks of miners in other threads is respectively 33.6, 33.8 and 33.9 bph when $TL = 800$ txps, and Fig. 7 (c) shows that it is 28.6, 29.3 and 30.2 bph when $TL = 8,000$ txps. The rate of final blocks of miners in the mixed strategy is respectively 33.6, 33.7 and 33.7 bph when $TL = 800$ txps and 28.6,

29.2 and 30.0 bph when $TL = 8,000$ txps.

Also, the rate of final blocks of the attacker is similar to or slightly above the one of the miners in the mixed strategy when $TL = 800$ txps, and significantly higher when $TL = 8,000$ txps. However, the rate of final blocks of miners in the same thread as the attacker gets significantly lower than the one of the miners in the mixed strategy when P_A increases to 20%, for any T, t_0^v and TL. For instance, when $T = 32$ threads and $t_0^v = 32$s, Fig. 7 (a) and (c) show that with P_A being 3%, 10% and 20%, the rate of final blocks of the attacker is respectively 34.3, 33.8 and 34.4 bph when $TL = 800$ txps and 29.8, 33.4 and 34.6 bph when $TL = 8,000$ txps, while the rate of final blocks of miners in the same thread as the attacker is respectively 33.1, 30.8, and 27.0 bph when $TL = 800$ txps, and 26.8, 25.4 and 21.9 bph when $TL = 8,000$ txps.

Second, for all tested combinations of T and t_0^v, the rate of final transactions of miners staying in other threads and of miners in the mixed strategy increases when the attacker's proportion of mining power P_A increases. For instance with $T = 32$ threads and $t_0^v = 32$s, Fig. 7 (b) shows that with P_A being 3%, 10% and 20%, the rate of final transactions of miners in other threads is respectively 8.7, 9.4 and 10.6 when $TL = 800$ txps, and Fig. 7 (d) shows that it is 62.0, 63.6 and 65.8 txps when $TL = 8,000$ txps. The rate of final transactions of miners in the mixed strategy is respectively 8.6, 9.2 and 10.3 txps when $TL = 800$ txps and 61.3, 62.2 and 64.1 txps when $TL = 8,000$ txps.

However, the rates of final transactions of the attacker and of miners of the same thread as the attacker quickly decrease when P_A increases, both when $TL = 800$ and $TL = 8,000$ txps, for any T and t_0^v. For instance, when $T = 32$ threads and $t_0^v = 32$s, Fig. 7 (b) and (d) show that with P_A being 3%, 10% and 20%, the rate of final transactions of the attacker is respectively 4.5, 2.1 and 1.2 when $TL = 800$ txps and 43.0, 19.9 and 11.3 txps when $TL = 8,000$ txps, while the rate of final transactions of miners in the same thread as the attacker is respectively 4.4, 1.9 and 1.0 txps when $TL = 800$ txps, and 39.0, 17.3 and 8.6 txps when $TL = 8,000$ txps.

5 Discussion

We defined the blockclique architecture to tackle the scaling of transaction throughput through the implementation of transaction sharding in a multithreaded directed acyclic block graph structure. This architecture separates transactions and blocks into multiple threads so that two blocks of transactions of different threads can be compatible even if created at the same time. We then defined a consensus rule so that nodes agree on which blocks become final: the thread incompatibility and the grandpa incompatibility rules specify which blocks are compatible with each other, and the blockclique rule specifies the best clique of compatible blocks. The blockclique architecture works in principle with any Sybil-resistant time-regulated random node selection mechanism such as Proof-of-Work or Proof-of-Stake, although parts of the consensus rule may need to be adapted to the specifics of the time regulation.

We provided an example implementation of the different components of the architecture in a Proof-of-Work setting, however those components may be replaced by other implementations. For example, there are probably other ways to define the grandpa incompatibility rule and the blockclique rule, as long as they allow an efficient synchronization between threads and consensus between nodes. Also, we gave a description of a possible reward scheme with an exponential decay and a bounded total supply, although any other reward scheme could be implemented, such as a long-tail reward scheme. The target of blocks can have any specification as long as two thread incompatible blocks have the same target so that nodes cannot manipulate their block target, and the target takes into account the block rate in all threads and is not specific to the block thread, in order to mitigate the potential attack of switching to a thread to mine until the target gets harder and then switching to an easier thread, which would be possible with a thread-specific target with Proof-of-Work.

Then, we have implemented an open-source simulation of the interaction of nodes in a random peer-to-peer network, that discover blocks, ask and send blocks to peers, and compute the blockclique based on the blocks they received, with a Proof-of-Work random node selection mechanism. We varied the number of threads T and the time between blocks t_0^v, the average network bandwidth U and latency L, and the number of nodes N in the network. We ran several trials for each set of parameters, with different random seeds and up to $3,600$ created blocks, and we measured the average transaction throughput, stale rate, and total transaction time.

The simulations first show that when we increase the number of threads from $T = 1$ (single-thread blockchain) to $T = 32$ threads, the average transaction throughput goes to $6,285$ or $6,997$ transactions per second, assuming an average upload bandwidth of nodes of respectively $U = 32$ or $U = 64$ Mbps. However, if we increase the number of threads T even more, then either the time between blocks or the finality parameter should also be increased to keep the same level of security, e.g. $\frac{F \times t_0^f}{T} \geq 64$s, but then the average transaction time will also increase. The total transaction time is the time needed for a new transaction to be included in a block, plus the confirmation time: the time for this block to become a final block. With a number of threads $T = 32$, a time between blocks $t_0^v = 32$s and a finality $F = 64$, the average time needed for a new transaction to be included in a block is about 41s, and the average confirmation time is about 85s, so that the average total transaction time is 126 seconds assuming an aver-

age upload bandwidth of 32 Mbps. Interestingly, even if for the high security of the blockclique architecture, the finality parameter is set to $F = 64$ blocks, for low-cost transactions one can wait less and still consider the transaction confirmed with high probability (but not final yet). For instance, waiting that the transaction is included in a block included in the blockclique and that the blockclique becomes 16 blocks ahead of any other clique not containing the transaction reduces the total transaction time from 2 minutes to about 1 minute.

In the case where the network is not saturated, for instance if the average load of transactions per second is about 100 txps, the blocks are on average much smaller and thus transmitted much faster, so that the stale rate is much lower and the transaction time is a bit faster. Therefore the architecture is adapted to and efficient in a wide range of transaction throughput. However, to increase even more the maximum transaction throughput we would need to assume a higher average network bandwidth, e.g. 128 Mbps or above. Indeed, our experiments show that the average bandwidth of nodes has a large impact on the resulting transaction throughput. In a network with $1,024$ nodes, the average throughput is $6,285$ txps with an average upload bandwidth of 32 Mbps, and goes up to $6,997$ txps with $U = 64$ Mbps, while the stale rate goes from 20.7% down to 7.0%. The average latency, however, has a lower impact on the throughput, in part because we implemented the trick of [19]: when a node discovers or receives a block, it first sends the block header to each of its successors in the network graph before verifying the block and its transactions, and only sends the full block when the verifications are done. If the average bandwidth improves on the long term, the block size can then be increased to some extent, e.g. to 2 MB (for $T = 32$ threads and $t_0^v = 32$s), but we may exceed the limits of fair computation and storage resources if we increase the block size too much, unless an increase of the average computation and storage resources matches the improvements of network connectivity.

As the network grows in terms of the number of nodes, other parameters held constant, our simulations show (up to $4,096$ nodes) that the transaction throughput slightly decreases with the number of nodes in the network. As the broadcasting speed of blocks in the network is limited by the network diameter (the higher the average distance between two nodes, the slower the broadcasting), and as the diameter is a logarithmic function of the number of nodes (in a random network), the transaction throughput should indeed decrease, and decrease more and more slowly as the network grows.

Those results together show that the blockclique architecture, through the parallel discovery of blocks every $t_0^v = 32$ seconds in $T = 32$ different threads, scales to a high transaction throughput of more than $6,000$ transactions per second, with a low average total transaction time of 2 minutes, even in large networks of nodes. Related work such as Elastico, OmniLedger or Zilliqa (see Sec. 2) claim that with their respective method, the transaction throughput increases with number of nodes, whereas in their architectures, this may only be true with a low number of nodes or a low transaction throughput because they will necessarily hit the limits of network transmission time at some point. However they do not provide any simulation regarding the crucial question of which transaction throughput can be reached depending on the number of nodes and the average network bandwidth and latency. In our experiments, we simulated a network with heterogeneous bandwidths, where nodes with a higher bandwidth send blocks to more peers than nodes with a lower bandwidth, and the maximum bandwidth of the nodes was two times the average bandwidth. In large networks, relay nodes can be implemented to quickly send blocks across continents, see for example the Falcon network in Bitcoin, such that the block transmission can be more efficient in practice.

In the blockclique architecture with a PoW random node selection mechanism, miners can choose one or several threads in which they will mine new blocks. We defined a default behavior: small miners mine in a single thread at a time and regularly move to a random thread. Miners with more resources can mine in several random threads at the same time and regularly change threads or even mine in all threads if their transaction processing and network capacity allow it. However, as miners choose their mining thread, they can deviate from this default behavior and try to unbalance the mining power in threads, either by coordinating with a group of miners to mine in a specific thread, or for a very large miner by mining in only one thread even if it has the majority of the mining power of that thread.

We simulated this imbalance attack depending on the mining power of an attacker (from 0.5% to 20%), the transaction load and the blockclique structure parameters: the number of threads T and the time between two blocks t_0^v. We measured the rate of final blocks and transactions that were obtained per percentage of the total mining power, for the different miners: the attacker, the miners in the attacker's thread, and the miners in the other threads. We also estimated the returns of a miner that would have used the default strategy of regularly changing to a random thread. The simulations show that when the mining power proportion of the attacker increases from 0.5% to 20%, the rate of final blocks of the attacker per percentage of total mining power slightly increases, while its rate of final transactions largely decreases, both with a non-saturated or saturated transaction load, whereas the rate of final blocks of honest miners that regularly change to a random thread slightly increase and their rate of final transactions also increases (see Fig. 7). For example, with $T = 32$ threads and $t_0^v = 32$ and a non-saturated transaction load $TL = 800$ txps, an attacker with 3%,

10% or 20% of the mining power respectively mines 34.3, 33.8 or 34.4 blocks per hour and per percentage of the total mining power and 4.5, 2.1 or 1.2 transactions per second and per percentage of the total mining power, while an honest miner with the default strategy mines 33.6, 33.6 or 33.7 blocks per hour and 8.6, 9.2 or 10.3 transactions per second. Indeed, when the mining power is unbalanced across threads, with one thread having much more mining power than the others, the number of blocks appearing in threads is also unbalanced: there are much more blocks in the thread of the attack and therefore those blocks have much less transactions than in other threads if the transaction load is not saturated ($TL = 800$ txps). However, the number of blocks per percentage of mining power is slightly higher for attackers because their blocks have slightly less chances to become stale as any two blocks of the same attacker will be compatible. When the transaction load is saturated ($TL = 8000$ txps), the blocks are larger, but still smaller in the attack's thread than in other threads, so the blocks of the attacker are broadcasted faster and thus have a lower stale rate than the blocks of the honest miners with the default strategy. For instance, when the mining power proportion of the attacker is 20%, the rate of final blocks of the attacker is 34.6 while the one of honest miners with the default strategy is 30.0, and the rate of final transactions of the attacker is 11.3 while the one of honest miners is 64.1.

The total return of miners is the sum of the block rewards and the transaction fees, so that the exact cost of this attack on the attacker ultimately depends on the relative value of transaction fees and block rewards. For instance, if the average fees from the transactions in a block is of the same order as the block reward, then an attacker with 20% of the total mining power will get about 30% less rewards than honest miners, and if the average fees from transactions are 10 times the average block rewards, then the same attacker will get about 75% less rewards. However in any case, the higher the attacker's mining power, the higher the number of final blocks and transactions and thus the rewards of honest miners. Indeed, the rate of final blocks and transactions of miners that would stay in the attacker's thread would be lower, but regularly changing to a random thread allows to mitigate the costs or this attack on honest nodes and to actually get returns that increase when the mining power of the attacker increases. We also simulated this attack with other structure parameters such as the number of threads and the time between blocks (see Appendix B), and we obtained similar results which show that the blockclique architecture is robust against imbalance attacks for a wide range of structure parameters. We simulated a single attacker with a very large proportion of the total mining power, however a group of attackers could join their forces to together mine in a same thread. In the case where those miners pool into the same facility, with almost zero latency between the different mining units, then the results of this attack are equivalent to a single attacker with a very large mining power, however if attackers are spread among different places, with an average latency between attackers, then attackers will mine incompatible blocks and their returns from this attack will be even lower.

With Proof-of-Work, the nodes can choose the thread in which they want to produce new blocks which makes the imbalance attack possible. However, a Proof-of-Stake mechanism could select for each block to be produced both a random stake and a random thread, so that nodes could not manipulate the thread in which their stake will allow them to produce a block. PoW mechanisms also have the drawback that nodes buy more computing power to get more rewards in a competition for the highest mining power proportion. For instance, in the Bitcoin network, the estimated current energy waste is somewhere between 1 GW and 10 GW for about 40×10^{18} hashes per second as of July 2018.

Another interesting use of blockchains is the processing of complex smart contracts, for example with the Ethereum platform [2]. In the blockclique however, transaction sharding requires that the contracts processed through a transaction in a particular thread can only reduce the balance of addresses from this particular thread. Also, the necessary forgetting of old blocks in a high-throughput architecture constrains the contracts to fit in few data (the "balance" of a contract), so that nodes can forget the transactions that created or interacted with the contract and only look at and evolve this stored data when executing new transactions. Finally, the contract must either be accessed from a single specific thread, or be able to operate in a multithreaded environment and resolve potential incompatibilities between interactions that happened at the same time in compatible blocks of different threads. The blockclique architecture as described here can thus be used to implement particular smart contracts that fit those constraints, such as Ethereum's ERC20, but can't implement any arbitrary smart contracts.

Our simulations have some limitations. First, the network could be more realistic, for instance we could simulate larger networks, and with a more realistic topology. Also, we did not simulate the transmission of unconfirmed transactions between nodes, which may consume as much bandwidth as block transmission. However, if nodes only download the transactions from the thread where they plan to mine (with PoW) or are selected to produce a block (with PoS), then the bandwidth necessary for sharing those unconfirmed transactions is much lower. Finally, we did not implement block targets and we assumed that all honest miners have the same constant mining power, but variations on the mining power or stake of nodes are expected to have little effects on the global transaction throughput and stale rate as nodes are choosing or selected to produce blocks in random threads.

6 Conclusion

The blockclique architecture combines three main ideas that together make scaling possible:

1) transaction sharding, where transactions are separated into multiple groups based on their input address so that transactions in two blocks of different groups created at the same time can be compatible,

2) a multithreaded directed acyclic block graph structure, where each block references one previous block of each thread, and

3) representing transactions as a value moved from one address to another and storing the balance of each address to allow forgetting old blocks.

The architecture specifies simple rules so that nodes agree on which blocks become final: the thread incompatibility rule states that two blocks of a thread cannot have the same parent in their thread which would be equivalent to a fork in a blockchain, the grandpa incompatibility rule pushes miners to take into account the blocks discovered in other threads, and the blockclique rule defines the blockclique as the clique of compatible blocks of maximum work sum, and if tie, of minimum hash sum.

In order to evaluate how the blockclique architecture scales transaction throughput, we have implemented an open-source simulation of the interaction of nodes in a random peer-to-peer network with a Proof-of-Work setting, that discover blocks, ask and send blocks to peers, and compute the blockclique based on the blocks they received. We have chosen parameters to use the maximum of today's fair storage capacity, network connectivity and computation capacity at the same time. For instance, processing about $8,000$ transactions per second and an upload bandwidth of 32 Mbps is still reasonable even for small nodes, and old blocks can be forgotten so that nodes may save only what their storage capacity allows. We then defined the block size BS so that about $8,000$ transactions per second can be processed, with e.g. $BS = 1$ MB for $T = 32$ blocks and $t_0^v = 32$s. Indeed, we reduced the amount of data included in transactions and blocks to the bare minimum, giving a transaction size of 127 B and a block header size of 175 B (with $T = 32$ blocks), so that a block of 1 MB can include $7,872$ transactions. With those settings, we then varied the number of threads T and the time between blocks t_0^v to evaluate different blockclique structure parameters, and we also experimented different ranges of average network bandwidth and latency starting from estimates of the connectivity in Bitcoin and Ethereum networks. We ran several trials for each set of parameters, with different random seeds and up to $3,600$ created blocks, and we measured the average transaction throughput, stale rate, and total transaction time.

The main result of our experiments is that the blockclique architecture, through the parallel discovery of blocks every $t_0^v = 32$ seconds in $T = 32$ different threads, scales to a high transaction throughput of more than $6,000$ transactions per second, with a low average total transaction time of 2 minutes, assuming a reasonable network connectivity with an average upload bandwidth of 32 Mbps and an average latency of 100 ms in a network of $1,024$ nodes, and a high-security finality parameter of $F = 64$ blocks in a Proof-of-Work setting. The blockclique architecture also removes the incentive for miners to pool, which protects decentralization, as one reward is distributed on average every second, and the decentralized network is secure against double-spending, replay attacks, imbalance attacks and selfish mining attacks.

Some related work study transaction sharding [5, 6, 7] but in the end only allow one chain of blocks, and some other work use a block graph structure [8, 9, 10], but do not shard transactions and therefore need complex strategies to resolve transaction incompatibilities. The blockclique is the first architecture to combine transaction sharding and a multithreaded directed acyclic block graph structure, building parallel threads of blocks and using a simple clique algorithm to find the best set of compatible blocks. With respect to the blockchain, the blockclique architecture thus extends the data structure to a multithreaded block DAG and adapts the Nakamoto consensus rule to multiple threads, however it is compatible with any Sybil-resistant time-regulated random node selection mechanism, such as Proof-of-Work or Proof-of-Stake.

Contributions

S.F. designed the architecture, wrote the source code, performed the experiments and wrote most of the paper. D.V. updated the consensus rule and provided critical feedback on previous versions of the architecture and results.

Acknowledgements

We thank Théo Segonds, Baptiste Busch and William Schueller for our discussions on blockchains and blockcliques, and for their comments on previous versions of this manuscript.

References

[1] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008.

[2] V. Buterin, "Ethereum: A next-generation smart contract and decentralized application platform," 2013.

[3] A. Kiayias, A. Russell, B. David, and R. Oliynykov, "Ouroboros: A provably secure proof-of-stake blockchain protocol," in *Annual*

International Cryptology Conference, pp. 357–388, Springer, 2017.

[4] L. Goodman, "Tezos: A self-amending crypto-ledger," 2014.

[5] L. Luu, V. Narayanan, C. Zheng, K. Baweja, S. Gilbert, and P. Saxena, "A secure sharding protocol for open blockchains," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pp. 17–30, ACM, 2016.

[6] E. Kokoris-Kogias, P. Jovanovic, L. Gasser, N. Gailly, and B. Ford, "Omniledger: A secure, scale-out, decentralized ledger," *IACR Cryptology ePrint Archive*, 2017.

[7] "The zilliqa technical whitepaper." Version 0.1.

[8] Y. Lewenberg, Y. Sompolinsky, and A. Zohar, "Inclusive block chain protocols," in *International Conference on Financial Cryptography and Data Security*, pp. 528–547, Springer, 2015.

[9] Y. Sompolinsky, Y. Lewenberg, and A. Zohar, "Spectre: A fast and scalable cryptocurrency protocol," *IACR Cryptology ePrint Archive*, 2016.

[10] Y. Sompolinsky and A. Zohar, "Phantom: A scalable blockdag protocol," *IACR Cryptology ePrint Archive*, 2018.

[11] I. Eyal, A. E. Gencer, E. G. Sirer, and R. V. Renesse, "Bitcoin-ng: A scalable blockchain protocol," in *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pp. 45–59, USENIX Association, 2016.

[12] Y. Sompolinsky and A. Zohar, "Secure high-rate transaction processing in bitcoin," in *International Conference on Financial Cryptography and Data Security*, pp. 507–527, Springer, 2015.

[13] S. Popov, "The tangle," 2017.

[14] A. Back, M. Corallo, L. Dashjr, M. Friedenbach, G. Maxwell, A. Miller, A. Poelstra, J. Timón, and P. Wuille, "Enabling blockchain innovations with pegged sidechains," 2014.

[15] J. Poon and T. Dryja, "The bitcoin lightning network: Scalable off-chain instant payments," *draft version 0.5*, vol. 9, p. 14, 2016.

[16] D. Schwartz, N. Youngs, A. Britto, *et al.*, "The ripple protocol consensus algorithm," *Ripple Labs Inc*, 2014.

[17] L. Baird, "Hashgraph consensus: Fair, fast, byzantine fault tolerance," 2016.

[18] T. Crain, V. Gramoli, M. Larrea, and M. Raynal, "(leader/randomization/signature)-free byzantine consensus for consortium blockchains," *arXiv preprint arXiv:1702.03068*, 2017.

[19] C. Decker and R. Wattenhofer, "Information propagation in the bitcoin network," in *IEEE Thirteenth International Conference on Peer-to-Peer Computing (P2P)*, pp. 1–10, IEEE, 2013.

[20] I. Eyal and E. G. Sirer, "Majority is not enough: Bitcoin mining is vulnerable," in *International conference on financial cryptography and data security*, pp. 436–454, Springer, 2014.

[21] A. E. Gencer, S. Basu, I. Eyal, R. van Renesse, and E. G. Sirer, "Decentralization in bitcoin and ethereum networks," *arXiv preprint arXiv:1801.03998*, 2018.

Appendix

A Markov Chain Model

In this section we model an attack where a group of nodes decides to switch to a clique that is $F-1$ blocks behind the blockclique. We assume that attackers have a total proportion of mining/staking power p, that the stale rate is s, and that nodes observe all blocks immediately without latency, such that when a block is created, all nodes will have the same blockclique and alternative cliques, and a global state can represent the number of blocks an alternative clique is behind the blockclique for all nodes. Fig. 8 shows a Markov chain model representing this global state of an alternative clique with respect to the blockclique. We assume that we start in state $1-F$, meaning that the alternative clique is $F-1$ blocks behind the blockclique.

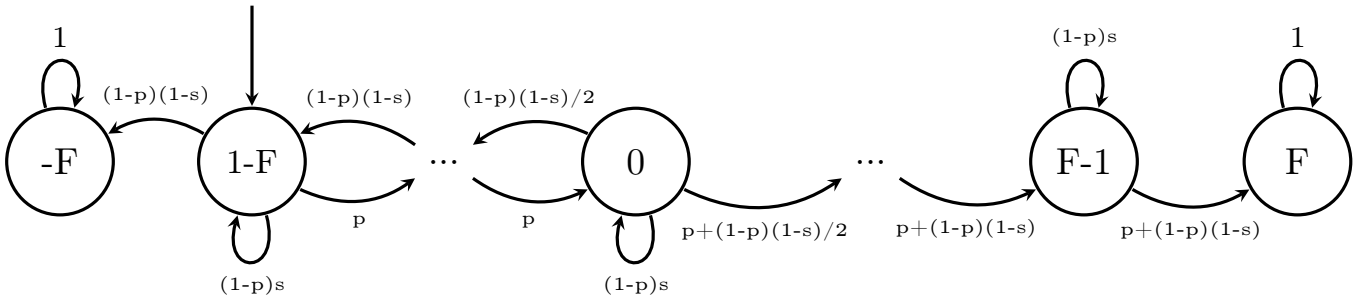


Figure 8: Markov chain representing the number of blocks of an alternative clique with respect to the blockclique.

When the alternative clique becomes F blocks behind the blockclique (state $-F$), it is removed and cannot compete with the blockclique anymore (absorbing state). When it gets the same size as the blockclique (state 0), then honest nodes, with $(1-p)(1-s)$ effective power, will either extend the alternative clique which will become the new blockclique and at some point be F blocks ahead of the previous blockclique (in absorbing state F), or extend the blockclique so that the alternative clique goes back 1 block behind.

We reorder the states of the Markov chain in the order $(1-F), \cdots, (F-1)$, $-F$, F, and write the transition matrix P as a function of Q, the probability of transitioning from a transient state to another and R, the probability of transitioning from a transient state to an absorbing state. Also, we call N the fundamental matrix, giving the expected number of times the chain is in a state i given that it started in state j, which is the sum for all k of the probability of transitioning from i to j in exactly k steps:

$$P = \begin{pmatrix} Q & R \\ \mathbf{0} & I_2 \end{pmatrix} \qquad N = \sum_{k=0}^{\infty} Q^k = (I_{2F-1} - Q)^{-1}$$

Finally, $B = NR$ is the probability to be absorbed in each absorbing state starting from a transient state i. Fig. 9 shows the probability to overtake the blockclique (to be absorbed in state F) starting from $1-F$.

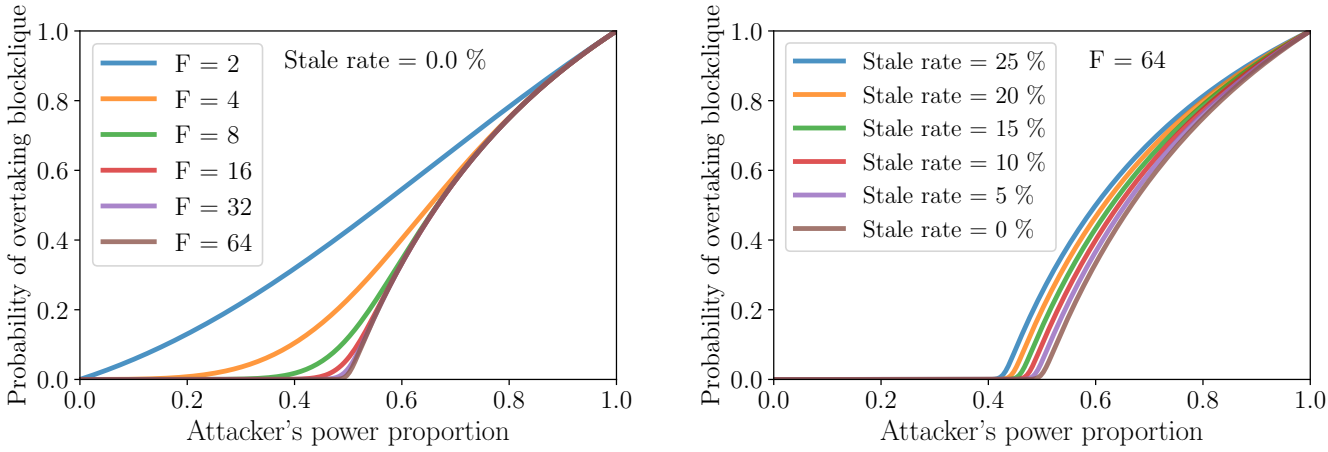
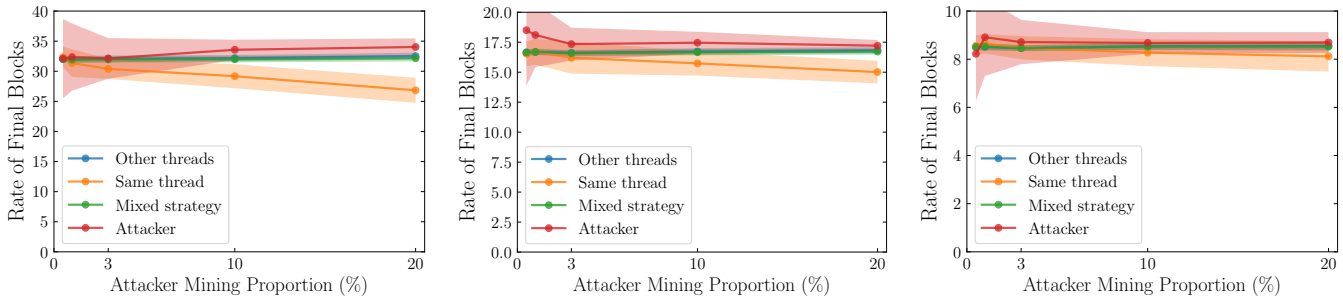
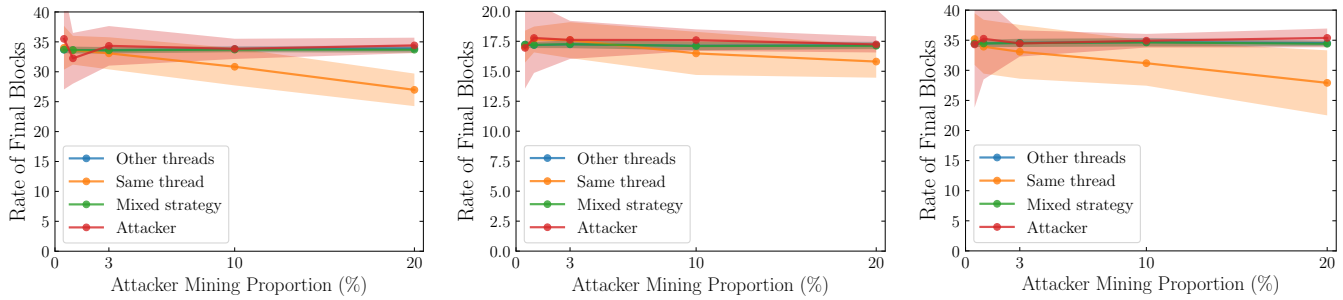


Figure 9: Probability for an alternative clique with $F-1$ blocks less than the blockclique to overtake it.

B Imbalance Attack

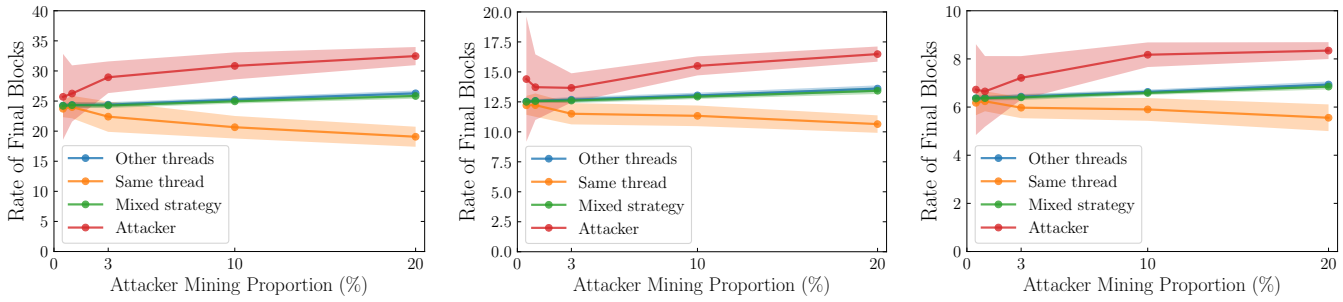


(a) $T = 16$ threads, $t_0^v = 16$s (b) $T = 16$ threads, $t_0^v = 32$s (c) $T = 16$ threads, $t_0^v = 64$s

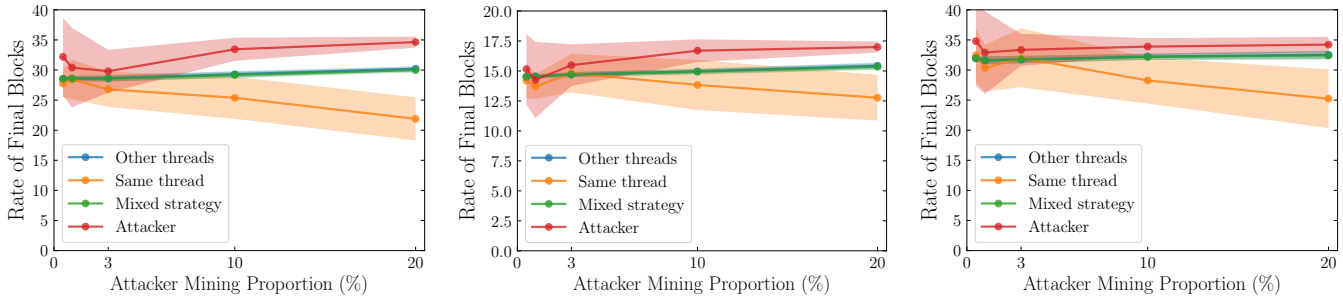


(d) $T = 32$ threads, $t_0^v = 32$s (e) $T = 32$ threads, $t_0^v = 64$s (f) $T = 64$ threads, $t_0^v = 64$s

Figure 10: Rate of final blocks of the different miners under the imbalance attack and a non-saturated transaction load $TL = 800$ txps, depending on the attacker's proportion of mining power, T and t_0^v.

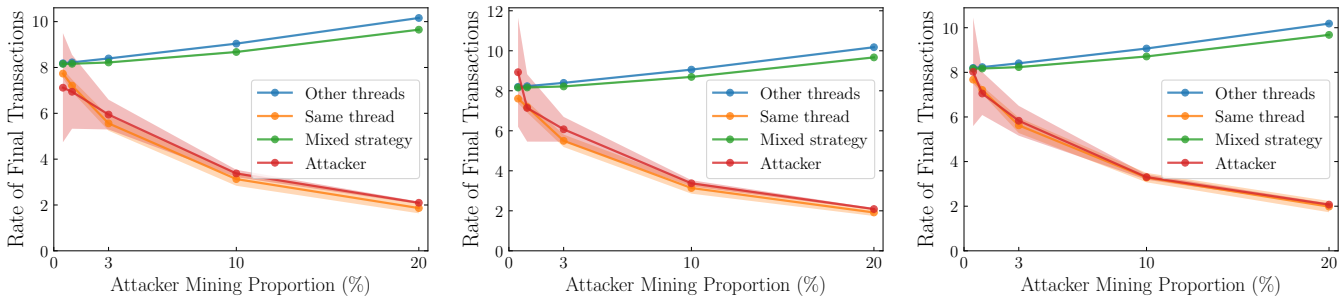


(a) $T = 16$ threads, $t_0^v = 16$s (b) $T = 16$ threads, $t_0^v = 32$s (c) $T = 16$ threads, $t_0^v = 64$s

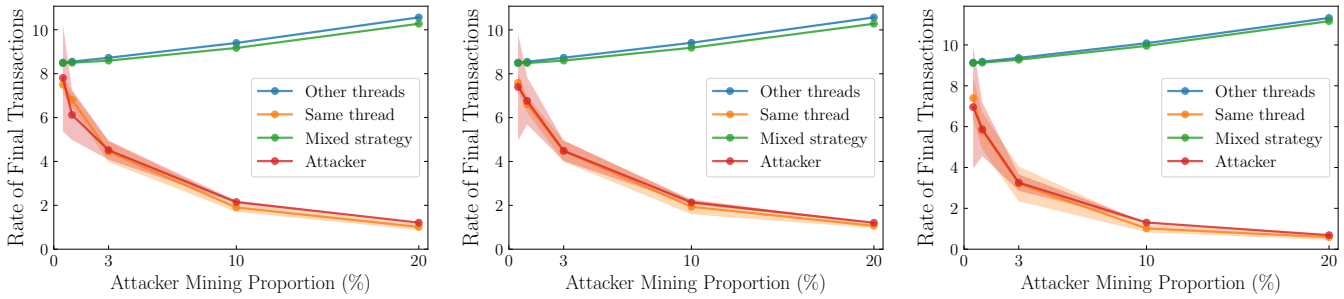


(d) $T = 32$ threads, $t_0^v = 32$s (e) $T = 32$ threads, $t_0^v = 64$s (f) $T = 64$ threads, $t_0^v = 64$s

Figure 11: Rate of final blocks of the different miners under the imbalance attack and a saturated transaction load $TL = 8,000$ txps, depending on the attacker's proportion of mining power, T and t_0^v.

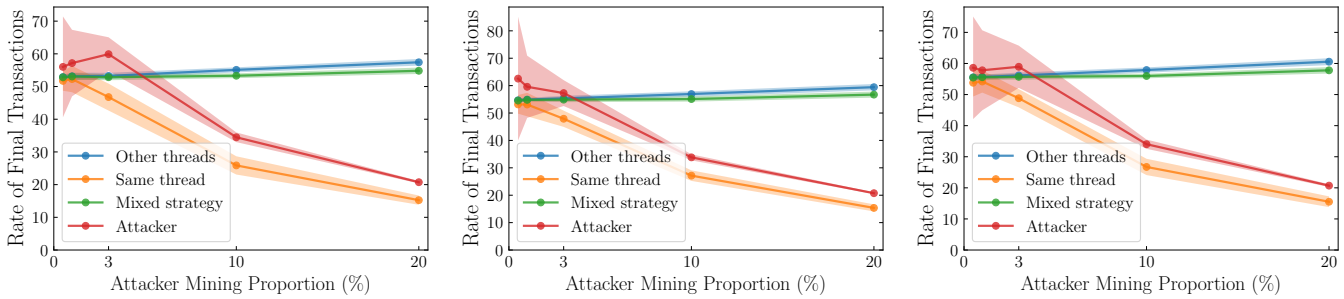


(a) $T = 16$ threads, $t_0^v = 16$s (b) $T = 16$ threads, $t_0^v = 32$s (c) $T = 16$ threads, $t_0^v = 64$s

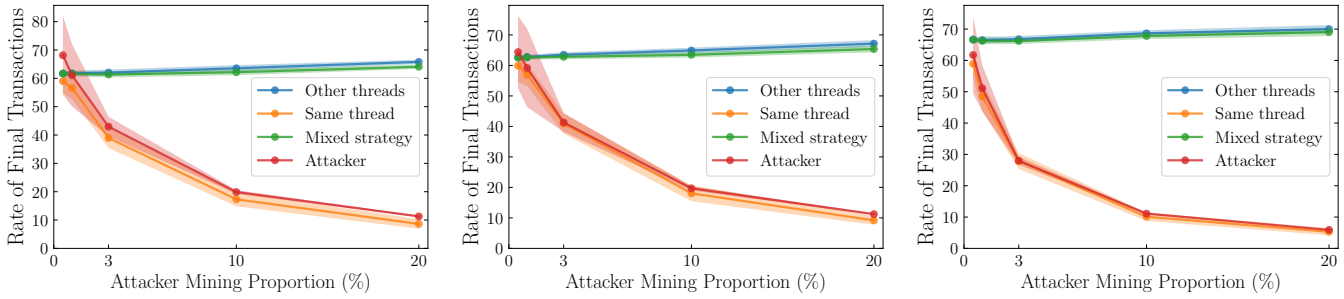


(d) $T = 32$ threads, $t_0^v = 32$s (e) $T = 32$ threads, $t_0^v = 64$s (f) $T = 64$ threads, $t_0^v = 64$s

Figure 12: Rate of final transactions of the different miners under the imbalance attack and a non-saturated transaction load $TL = 800$ txps, depending on the attacker's proportion of mining power, T and t_0^v.



(a) $T = 16$ threads, $t_0^v = 16$s (b) $T = 16$ threads, $t_0^v = 32$s (c) $T = 16$ threads, $t_0^v = 64$s



(d) $T = 32$ threads, $t_0^v = 32$s (e) $T = 32$ threads, $t_0^v = 64$s (f) $T = 64$ threads, $t_0^v = 64$s

Figure 13: Rate of final transactions of the different miners under the imbalance attack and a saturated transaction load $TL = 8,000$ txps, depending on the attacker's proportion of mining power, T and t_0^v.