Blockchain using Proof-of-Interaction

Jean-Philippe Abegg, Quentin Bramas, Thomas Noel

Blockchain using Proof-of-Interaction

Jean-Philippe Abegg[1,2] Quentin Bramas[1] Thomas Noël[1]

[1] ICUBE, University of Strasbourg, France
[2] Transchain, Strasbourg, France
jp.abegg@transchain.fr

Abstract

Proof-of-Work is originally a client-side puzzle proposed to prevent spam or denial of service attacks. In 2008, Satoshi Nakamoto used it as an election mechanism (or equivalently, to replace a centralized time server) in the first Blockchain: Bitcoin. In the same year, another spam prevention algorithm was proposed, based on a guided-tour puzzle, but received only little attention.

The main motivation of our work is to see if a Blockchain protocol can use the guided-tour puzzle like Bitcoin uses Proof-of-work.

In this paper we extend the guided tour puzzle to a new Puzzle called Proof-of-Interaction and we show how it can replace, in the Bitcoin protocol, the Proof-of-Work algorithm. We show that it uses a negligible amount of computational power compared to Bitcoin, and scales very well in term of number of messages. We analyze the security of our protocol and show that it is not subject to selfish mining. However, our protocol currently works only when the nodes in the network are known, but we discuss how this assumption could be weakened in future work.

1 Introduction

A Blockchain is a Distributed Ledger Technology (DLT) *i.e.*, a protocol executed by a set of nodes to maintain a data-structure where data can only be appended in blocks. It is maintained in a distributed manner by many participants, who may not trust each-other, and some of which can be faulty or malicious. In order for this data-structure to be consistent among the participants, a protocol is used to ensure that every one agrees on the next block that is appended into the Blockchain.

The most famous example of such protocol, Bitcoin [18], uses the Proof-of-Work to elect a single participant that is responsible for appending the next block. In more details, Proof-of-work is a client-puzzle that is executed by all the nodes in the network. Finding a solution of the puzzle requires a large amount of computational power but is easily verifiable. The first node that finds a solution is the one that is allowed to append a block to the chain of block. The difficulty of the puzzle increases with the total computational power of the network in order to limit the chance of having multiple concurrent elected nodes. This implies that the total power consumption increases linearly with the total computational power of the participants. According to the latest estimates, the Bitcoin network consumes more than the Czech Republic [7, 11] (and just account for less than 70% of all the Proof-of-work based Blockchains).

There have been many attempts to avoid using Proof-of-work based agreement, but usually adding other constraints [22] (*eg.*, small number of nodes, hardware prerequisite, new security threats).

In this paper, we propose to use a new client-puzzle called Proof-of-Interaction to define a new energy-efficient Blockchain protocol.

Related Work. Proof-of-work [4] (PoW) is a method initially intended for preventing spamming attacks. It was then used in the Bitcoin protocol [18] as a way to prove that a certain amount of time has passed between two consecutive blocks. Another way to see the aim of the Proof-of-work in the Bitcoin protocol is as a leader election mechanism, to select who is responsible for writing the next block in the blockchain. This leader election has several important properties, including protection against Sybil attacks [9] and against denial-of-service [16]. Also, it has a small communication complexity. However, the computational race consumes a lot of energy. The majority of current Blockchain protocols uses Proof-of-Work, with different hashing functions [17].

In 2012, S. King and S. Nadal [14] proposed the Proof-of-Stake (PoS), an alternative for PoW. This leader election mechanism requires less computational power but has security issues [12, 5] (*eg.*, Long range attack and DoS). Intel proposed another alternative to PoW, the Proof-of-Elapsed-Time (PoET) [1]. This solution requires Intel SGX as a trusted execution environment. Thus, Intel becomes a required trusted party to make the consensus work, which might imply security concerns [8] and is against blockchain idea to remove third parties. Other mechanisms where proposed such as Proof-of-Activity and Proof-of-Importance [3], which are hybrid protocols or protocols using properties from the network itself.

Vote-based protocols refer to the family of Byzantine Agreement protocols, such as PBFT [6]. Such protocols do not use client-puzzle, hence are energy-efficient. They can handle a large amount of transactions but must be executed in a known network, and do not scale well with the number of nodes due to their communication complexity.

The previous paper most related to our work was presented just prior the publication of Bitcoin in 2008, by M. Abliz and T. Znati [2]. They proposed *A Guided Tour Puzzle for Denial of Service Prevention*, which is another spam protection algorithm. This mechanism has not yet been used in the Blockchain context, and is at the core of our new Proof-of-Interaction. The idea was that, when a user wants to access a resource in a server that is heavily requested, the server can ask the user to perform a tour of a given length in the network. This tour consists of accessing randomly a list of nodes, own by the same provider as the server. After the tour, a user can prove to the server that it has completed the task and can then retrieve the resource. The way we generate our tour in our Proof-of-Interaction is based on the same idea. We generalized the approach of M. Abliz and T. Znati to work with multiple participants, and we made the tour length variable.

Contributions. The contribution of this paper is twofold. First, we propose a better alternative to Proof-of-Work, called Proof-of-Interaction, which requires negligible computational power. Second, we show how it can be used to create an efficient Blockchain protocol that is resilient against selfish mining, but assumes for now that the network is known.

Paper Structure. Section 2 presents the model and illustrates the problem with naive approaches. This also helps to understand how our protocol is built. Then we present the Proof-of-Interaction protocol, that could be used outside of the Blockchain context. Then, in Section 4 we explain how we use this proof mechanism to create a Blockchain protocol. In Section 5, we analyze the security properties of our protocol. Finally, we conclude and discuss possible extensions in Section 6.

2 Preliminaries

2.1 Model

The network, is a set $\mathcal{N}$ of n nodes that are completely connected. Each node has a pair of private and public cryptographic keys. Nodes are uniquely identified by their public keys (*i.e.*, the association between the public keys and the nodes is common knowledge). Each message is signed by its sender, and a node cannot fake a message signed by another (non-faulty) node.

We denote by $\texttt{sign}_u(m)$ the signature by node u of the message m, and $\texttt{verif}_u(s, m)$ the predicate that is true if and only if $s = \texttt{sign}_u(m)$. For now, we assume the signature algorithm

is a deterministic one-way function that depends only on the message m and on the private key of u. This assumption might be very strong as, with common signature schemes, different signature could be generated for the same message, but there are ways to remove this assumption by using complex secret generation and disclosure schemes, not discussed in this paper, so that each signature is in fact a deterministic one-way function. The function H is a cryptographic, one-way and collision resistant, hash function [19].

As for the Bitcoin protocol, we assume the communication is partially synchronous *i.e.*, there is a fixed, but unknown, upper bound Δ on the time for messages to be delivered.

The size of a set S is denoted with $|S|$.

2.2 Guided Tour

The guided tour defined by M. Abliz and T. Znati [2] can be summarized as follow. When a resource server is under DOS attack, it responds to a given request by a random seed hash h_0, a set S of n servers and a length L. The client has to solve a puzzle in order to complete its request to the resource server. To solve the puzzle, the client makes L requests to the servers in S in a specific order. The index, in S, of the first server to request is deduced from h_0. Let $i_0 \in [0, n-1]$ such that $i_0 \equiv h_0 \mod n$. Then, the client sends message h_0 to the i_0-th server in S. The server responds with hash h_1. Then then client computes $i_1 \in [0, n-1]$ such that $i_1 \equiv h_1 \mod n$, and sends message h_1 to the i_1-th server in S, and so on. This continues until hash h_L is obtained. h_L is a proof that the tour as been completed, and is sent to the resource server to obtain the requested resource. Thanks to a secret shared among all the servers, the resource server is able to check that hash h_L is indeed the expected proof for the initial seed h_0. This idea is interesting because the whole tour depends only on the initial value, and cannot be performed in parallel because each hash h_i cannot be found until h_{i-1} is known. We then present a naive approach on how it can be used as a distributed client-puzzle.

2.3 Naive Approach

We give here a naive approach on how asking participants to perform a tour in the network can be used as a leader election mechanism to elect the node responsible for appending the next block in a Blockchain.

When a node u_0 wants to append a block to the blockchain, it performs a random tour of length L in the network retrieving signatures of each participants it visits. The first node u_1 to visit is the hash of the last block $h_0 = \mathit{last\_block\_hash}$ of the blockchain modulo n (if we order nodes by their public keys, the node to visit is the i-th with $i = h_0 \mod n$). u_1 responds with the signature $s_1 = \mathtt{sign}_{u_1}(h_0)$. The hash $h_1 = H(s_1)$, modulo n, gives the second node u_2 to visit, and so on. This idea is similar to the guided tour of M. Abliz and T. Znati [2], and here the whole tour depends only the hash of the last block. Given h_0, anyone can verify that the sequence of signatures $(s_1, s_2, \ldots, s_L)$ is a proof that the tour has been properly performed. If each node in the network performs a tour, the first node to complete its tour is elected broadcast its block, containing the proof, to the other nodes to announce it.

However, here, each node has to perform the same tour, which could be problematic. An easy fix is to select the first node to visit, not directly using the hash of the last block, but also based on the signature of the node initiating the tour, $h_0 = H(\mathtt{sign}_{u_0}(\mathit{last\_block\_hash}))$. Now, given h_0, the sequence of signatures $(\mathtt{sign}_{u_0}(h_0), s_1, s_2, \ldots, s_L)$ proves that the tour has been properly performed by node u_0. Each tour, performed by a given node, is unique, and a node cannot compute the sequence of signature other than by actually asking each node in the tour to sign a message. Indeed, the next hop of the tour depends on the current one.

Here, one can see that it could be a good idea to also make the tour dependent on the content of the block node u_0 is trying to append. Indeed, using only the last block to generate a new proof does not protect the content of the current block, *i.e.*, the same proof can be used to create two different blocks. To prevent this behavior, we can assume that $h_0 = H(\mathtt{sign}_{u_0}(\mathit{last\_block\_hash}) \cdot M)$ ($\cdot$ being the concatenation operator) where M is a hash of the content of the block node u_0

is trying to append. In practice, it is the root of the Merkel tree containing all the transactions of the block. Here, the proof is dependent on the content of the block, which means that if the content of the block changes the whole proof needs to be computed again.

From there, we face another issue. Each node performs a tour of length L, so each participant will be elected roughly at the same time, creating a lots of forks. To avoid this, we can make the tour length variable. We found two ways to do so. The first one is not to decide on a length in advance, and perform the tour until the hash of k-th signature is smaller than a given target value, representing the difficulty of the proof. In this way, every interaction with another node during a tour can be seen as a tentative to find a good hash (like hashing a block with a given nonce in the PoW protocol). The target value can be selected so that the average length of the tour is predetermined. However doing so, since the proof does depend on the content of the block, u_0 can change the content of the block, by adding dummy transactions for instance, so that the tour stops after one hop[1]. The other way to make the tour length variable is to use a cryptographic random number generator, seeded with $\texttt{sign}_{u_0}(last\_block\_hash)$, to generate the length L. Doing so, the length depends only on u_0 and on the previous block. Then a tour of length L is performed as usual.

To complete the scheme, we add other information to the message sent to the visited node so that they can detect if we try to prove different blocks in parallel. We also make u_0 sign each response before computing the next hop, so that the tour must pass through u_0 after each visit. Finally, we will see why it is important to perform the tour, not using the entire network, but only a subset of it.

3 The Proof-of-Interaction

In this section we define the most important piece of our protocol, which is, how a given node of the network generates a proof of interaction. Then, we will see in the next section how this proof can be used as an election mechanism in our Blockchain protocol.

3.1 Algorithm Overview

We present here two important algorithms. One that generates a Proof-of-Interaction (PoI), and one that checks the validity of a given PoI.

Generating a Proof-of-Interaction. Consider we are a node $u_0 \in \mathcal{N}$ that wants to generate a PoI. Given a fixed *dependency* value denoted d, the user u_0 wants to prove a *message* denoted m. The user has no control over d but can chose any message to prove.

The signature by u_0 of the dependency d, denoted $s_0 = \texttt{sign}_{u_0}(d)$, is used to generate the subset S of $n_S = \min(20, n/2)$ nodes to interact with

$$S = \{S_0, S_2, \ldots, S_{n_S-1}\} = \texttt{createServices}\,(\mathcal{N}, s_0)\,.$$

S is generated using the pseudo-random Algorithm `createServices`, and depends only on d and on u_0. From s_0, we also derive the length of the tour $L = \texttt{tourLength}(D, s_0)$, where D is a probabilistic distribution that corresponds to the difficulty parameter. `tourLength` is a random number generator, seeded with s_0 that generates a number according to D. Using D one can easily change the average length of the tour for instance.

Now u_0 has to visit randomly L nodes in S to complete the proof, as illustrated in Figure 1. To know what is the first node u_1 we have to visit, we first hash the concatenation of s_0 with m to obtain $h_0 = H(s_0 \cdot m)$. This hash (modulo $|S|$) gives the index i in S of the node we have to visit, $i \equiv h_0 \mod |S|$. So we send the tuple (h_0, d, m) to node $u_1 = S_i$, which responds by signing the concatenation, $s_1 = \texttt{sign}_{u_1}(h_0 \cdot d \cdot m)$.

[1]There are some ways to limit this attack, but we believe it will remain an important attack vector

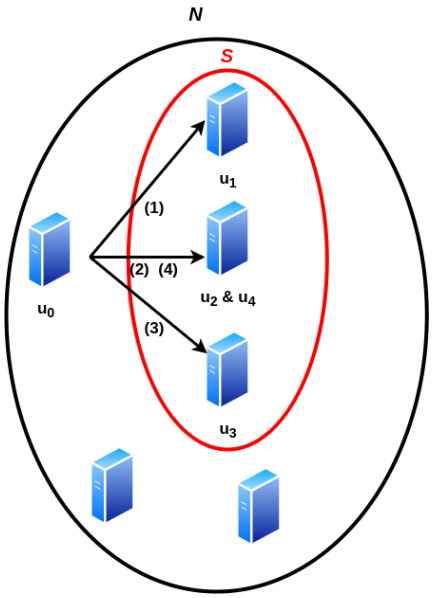

Figure 1: u_0 interacts randomly with a subset S of the nodes

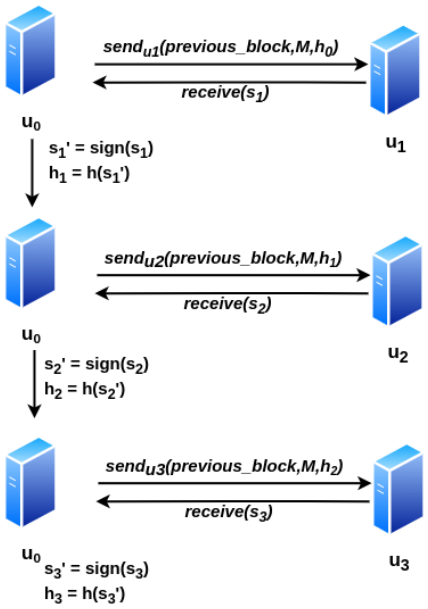

Figure 2: u_0 interacts with a sequence of nodes to construct a PoI. In this example, the dependency is the hash of the previous block.

To know what is the second node u_2 we have to visit, we sign and hash the response from u_1 to obtain $h_1 = H(\texttt{sign}_{u_0}(s_1))$, so that $u_2 = S_j \in S$ with $j = h_1 \mod |S|$. Again, we send the tuple (h_1, d, m) to u_2, which responds by signing the concatenation, $s_2 = \texttt{sign}_{u_2}(h_1 \cdot d \cdot m)$. We sign and hash the response from u_1 to obtain $h_2 = H(\texttt{sign}_{u_0}(s_2))$ and find the next node we have to visit, and so on (see Figure 2). This continues until we compute $\texttt{sign}_{u_0}(s_L)$, after the response of the L-th visited node.

The *Proof of Interaction* (PoI) with dependency d of message m by node u_0 and difficulty D is the sequence

$$(s_0, s_1, \texttt{sign}_{u_0}(s_1), s_2, \texttt{sign}_{u_0}(s_2), \ldots, s_L, \texttt{sign}_{u_0}(s_L)).$$

Checking a Proof-of-Interaction. To check if a PoI $(s_0, s_1, s'_1 \ldots, s_k, s'_k)$ from user u, is valid for message m, dependency d and difficulty D, one can first check if s_0 is a valid signature of d by u_0. If so, we can obtain the set $S = \texttt{createServices}(\mathcal{N}, s_0)$ of interacting nodes, the length $L = \texttt{tourLength}(D, s_0)$, and the hash $h_0 = H(s_0 \cdot m)$. From h_0 and S, we can compute what is the first node u_1 and check if s_1 is a valid signature from u_1 of $(h_0 \cdot d \cdot m)$, and if s'_1 is a valid signature of s_1 from u_0. Similarly, one can check all the signatures until s'_k. Finally, if all signatures are valid, and $k = L$, the PoI is valid.

3.2 Algorithm Details

The pseudo code of our algorithms are given below.

The algorithm `createServices` is straightforward. We assume that we have a random number generator (RNG) — defined by the protocol hence the same for all the nodes in the network — that we initialize with the given seed. The algorithm then shuffles the input array using the given random number generator. Finally, it simply returns the first n_S elements of the shuffled array.

The main part of the algorithm `generatePoI` consists in a loop, that performs the L interactions. The algorithm requires that each node in the network is executing the same algorithm (it can tolerates some faulty nodes, as explained later). The end of the algorithm shows what is

Algorithm createServices: create a pseudo-random subset of nodes

Input: N, the set of nodes
h, a seed
Output: S, a subset of nodes

$RNG.seed(h)$
$S \leftarrow \texttt{shuffled}(N, RNG)$
$S \leftarrow S.slice(0, n_S)$
return S

executed when a node receives a message from another node. The procedure `checkMessage` may depends on what the PoI is used for. In our context, the procedure checks that the nodes that interacts with us does not try to create multiple PoI with different messages, and use the same dependency as everyone else. We will see in details in the next section why it is important.

Algorithm generatePoI: Program executed by u_0 to generate the PoI

Input: d, the dependency (hash of last block of the blockchain)
m, the message (root of the merkle tree of the new block)
D, difficulty of the PoI
N, the set of nodes in the network
Output: P, a list of signatures $\{s_0, s_1, s'_1, s_2, s'_2, \ldots, s_k, s'_k\}$

$P \leftarrow [\,]$
$s_0 \leftarrow \texttt{sign}_{u_0}(d)$
$S \leftarrow \texttt{createServices}(N, s_0)$
$L \leftarrow \texttt{tourLength}(D, s_0)$
$P.append(s_0)$
$current\_hash \leftarrow H(s_0 \cdot m)$
for L *iterations* **do**
  $next\_hop \leftarrow current\_hash \% |S|$
  $s \leftarrow send_{S_{next\_hop}}(current\_hash, d, m)$
  $P.append(s)$
  $s \leftarrow \texttt{sign}_{u_0}(s)$
  $P.append(s)$
  $current\_hash \leftarrow H(s)$
return P

When Receive (h, d, m) from u **do**
if *checkMessage*(u, h, d, m) **then**
  Reply $\texttt{sign}_{u_0}(h \cdot d \cdot m)$

The algorithm `checkPoI` that checks the validity of a PoI is checking that each signature from the proof is valid and respects the proof generation algorithm.

Proof-of-Interactions Properties. Now we show that the Proof-of-Interaction has several properties that are awaited by client-puzzle protocols [21].

Computation guarantee: The proof can only be generated by making each visited node sign a particular message in the correct order. The sequence of visited node depends only on the initiator node, on the dependency d, and on the message m, and cannot be known before completing the tour. Furthermore, a node knows the size of his tour before completing it, which means that the node knows before doing his tour how much messages it needs to exchange and how much signatures it will do to have a correct proof.

Algorithm checkMessage: Check the message received from node u

Input: u, the sender of the request
h, difficulty of the PoI
d, the dependency (hash of last block of the blockchain)
m, the message (root of the merkle tree of the new block)
Output: whether to accept or not the request

```
1  if d is the hash of the latest block of one of the longest branches then
2      if Received[(u, d)] exists and is not equal to m then
3          penalties (u)
4          return false
5      Received[(u, d)] = m
6      return true
7  else
8      if unknown d then
9          Ask block d
10         return false
```

Algorithm checkPoI: Program executed by anyone to check the validity of a PoI

Input: P, a proof-of-interaction
u, creator of the proof
d, the dependency (hash of last block of the blockchain)
m, the message (root of the merkle tree of the new block)
D, difficulty of the PoI
N, the set of nodes in the network
Output: whether P is a valid PoI or not

```
1  if not verif_u(P[0],d) then
2      return false
3  S ← createServices(N, P[0]);
4  L ← tourLength(D, P[0]);
5  if L * 2 + 1 ≠ |P| then
6      return false
7  current_hash ← H(P[0] · m);
8  for i = 0; i < L; i++ do
9      next_hop ← current_hash%|S|;
10     if not verif_{S_next_hop}(P[2 * i + 1], current_hash · d · m) then
11         return false
12     if not verif_u(P[2 * i + 2], P[2 * i + 1]) then
13         return false
14     current_hash ← H(P[2 * i + 2]);
15 return true
```

Non-parallelizability: A node cannot compute a valid PoI for a given dependency d and message m in parallel. Indeed, in order to know what is the node of the i-th interaction, we need to know h_{i-1}, hence we need to know s_{i-1}. s_{i-1} is a signature from u_{i-1}. So we can interact with u_i only after we receive the answer from u_{i-1} *i.e.*, interactions are sequential.

Granularity: The difficulty of our protocol is easily adjustable using the parameter D. The expected time to complete the proof is $2 \times mean(D) \times Com$ where Com is the average duration of a message transmission in the network, and $mean(D)$ is the mean of the distribution D.

Efficiency: Our solution is efficient in terms of computation for all the participants. The generation of one PoI by one participant requires $mean(D)$ hashes and $mean(D)$ signatures in average for the initiator of the proof, and $mean(D)/n$ signatures in average for another node in the network. The verification requires $2D+1$ signature verification and $mean(D)$ hashes in average. The size of the proof is also linear in the difficulty, as it contains $2mean(D)+1$ signatures.

4 Blockchain Consensus Using PoI

In this section we detail how we can use the PoI mechanism to build a Blockchain protocol. The main idea is to replace, in the Bitcoin protocol, the Proof-of-work by the Proof-of-interactions, with some adjustments. We prove in the next section that it provides similar guarantees to the Bitcoin protocol.

Block Format. First, like in the Bitcoin protocol, transactions are stored in blocks that are chained together by including in each block, a field containing the hash of the previous block. In Bitcoin, a block includes a nonce field so that the hash of the block is smaller than a target value (hence proving that computational power has been used) whereas in our protocol, the block includes a proof of interaction where the dependency d is the hash of the previous block, and the message m is the root of the Merkel tree storing the transactions of the current block. Like for the transactions, the block header could contains only the hash of the PoI, and the full proof can be stored in the block data, along with the sequence of transactions.

Block Generation. Now we explain how the next block is appended in the blockchain. Like in Bitcoin, each participant gathers a set of transactions (not necessarily the same) and when the last block is received, wants to append a new block to the blockchain. To do so, each node tries to generate a PoI with the hash of the last block as dependency d, the root of the Merkel tree of the transactions of their own block as message m, and using the last block difficulty D. We assume the difficulty D is characterized by its mean value $mean(D)$, which is the number that is stored into the block. Like in Bitcoin, the difficulty can be adjusted every given period, depending on the time it takes to generate the last blocks.

Participants have no choice over d so the length of their tour, and the subset S of potential visited nodes is fixed for each participants (one can assume that it is a random subset). Each participant is trying to complete its PoI the fastest as possible, and the first one that completes it, has a valid block. The valid block is broadcasted into the network to announce to everyone that one have completed a PoI for its new block. When a node receives a block from another node, it checks if all the transactions are valid and then checks if the PoI is valid. If so, it appends the new block to its local blockchain and starts generating a PoI based on this new block.

First, one can see that this could lead to forks, exactly like in the Bitcoin protocol, where different part of the networks try to generate PoI with different dependencies. Thus, the protocol dictates that only one of the longest chain should be used as a dependency to generate a PoI. This is defined in the procedure `checkMessage`. When a node receives a message from another node, it first checks if the dependency matches the latest block of one of the longest chain. If not, the request is ignored.

Incentives. Like in Bitcoin, we give incentives to nodes that participate to the protocol. The block reward (that could be fixed, decreasing over time, or just contains the transactions fees) is evenly distributed among all the participants of the PoI of the block. This implies that, to maximize their gain, nodes should answer as fast as possible to all the requests from the other nodes currently generating their PoI, to increase their chance of being part of the winning block.

Also, it means that we do not want to answer a request for a node that is not up to date *i.e.*, that is generating a PoI for a block for which there is already a valid block on top, or for a block in a branch that is smaller than longest one.

Preventing Double-Touring Attacks. What prevents a node to try to generate several PoI using different variation of its block? If a node wants to maximize its gain (without even being malicious, but just rational) it can add dummy transactions to its current block to create several versions of it. Each version can be used to initiate the generation of a PoI using different tours. However, he has to send the message m every times he interacts with another node. If the length of the tour is long enough, the probability that two different tours intersect is very high. In other words, a node that receives two messages from the same node, with the same dependency d, but different values of m will raise the alarm. To prevent *double-touring*, it is easy to add an incentive to discourage nodes from generating several blocks linked to the same dependency. To do so, we assume each participant has locked a certain amount of money in the Blockchain, and if a node u has a proof that another node has created two different blocks with the same dependency (*i.e.*, previous block), then the node u can claim as reward the locked funds of the cheating node. In addition, it can have other implications such as the exclusion of the network. We assume that the potential loss of being captured is greater than the gain (here the only gain would be to have a greater probability to append its own block).

Difficulty Adjustment. The difficulty could be adjusted exactly like in Bitcoin. The goal is to chose the difficulty so that the average time B to generate a block is fixed. Here, the difficulty parameter D gives a very precise way to obtain a delay B between blocks and to limit the probability of fork at the same time. If Com denotes the average duration of a transmission in the network, then we want the expected shortest tour length among the participants to be $\lceil B/Com \rceil$.

For instance, it is known that the average minimum of n independent random variables uniformly distributed on the interval (a, b) is

$$\frac{b + na}{n + 1}.$$

Thus, if D is the uniform distribution between 1 and $\lceil B/Com \rceil (n+1) - 1$, then the length of the shortest tour among all the participants will be $\lceil B/Com \rceil$ in average.

Every given period (*eg.*, 2016 blocks as in Bitcoin), the difficulty could be adjusted using the duration of the last period (using the timestamps included in each block) to take into account the possible variation of Com, so that the average time to generate a block remains B.

Communication Complexity. A quick analysis shows that each node sends messages sequentially, one after receiving the answer of the other. At the same time, it answers to signature requests from the other nodes. In average, a node is part of n_S tours. Hence the average number of messages per unit of time is constant *i.e.*, $n_S + 1$ every Com. Then, the total amount of messages, per unit of time, in the whole network is linear in n.

5 Security

This section discusses about common security threats and how our PoI-based Blockchain handles them. We assume that honest nodes will always follow our algorithms but an attacker can have arbitrary behavior, while avoiding receiving any penalty (which could remove him from the network). We assume that an attacker can eavesdrop every messages exchange between two nodes but he can not change them. Also, assume that an attacker A cannot forge messages from another honest node B.

Crash Faults. A node crashes when it completely stops its execution. The main impact is that it does not respond to the sign requests of other nodes. This can be an issue because at each step of the PoI generation, the initiator node could wait forever the response of a crashed node. Crashed nodes are handled by the fact that a node only has to interact with a subset S

of the whole network $\mathcal{N}$, computed using the service creation function, `createServices`. Hence, if a node crashes, only a fraction of the PoI that are being generated will be stuck waiting for it. All the nodes whose Service sets S do not contains crash faults are able to generate their PoI entirely. Since each set S is of size $n_S = \min(n/2, 20)$, we have that, if half of the nodes crash, the probability a given set S contains a crashed node is $1 - \left(\frac{1}{2}\right)^{n_S}$. So that the probability p that at least one set S contains only correct nodes is

$$p = 1 - \left(1 - \left(\frac{1}{2}\right)^{n_S}\right)^n$$

One can see that the probability p tends quickly (exponentially fast) to 1 as n tends to infinity. For small values of n, the probability is greater than a fixed non-null value. In the rare event that all the sets S contain at least a crashed node, then the protocol is stuck until some crashed nodes reboot and are accessible again.

Finally, we recall that honest nodes are incentivized to answer, because they get a reward when they are included in the next block's PoI. Hence, honest nodes will try be back again as fast as possible.

Selfish mining. Selfish mining [10] is an attack where a set of malicious nodes collude to waste honest nodes resources and get more reward. It works as follow. Once a malicious node finds a new block, it only shares it with the other malicious nodes. All malicious nodes will be working on a private chain without revealing their new block, so that honest nodes are working on a smaller public branch *i.e.*, honest nodes are wasting resources to find blocks on a useless branch. When honest nodes find a block, the malicious nodes might reveal some of their private blocks to discard honest blocks and get the rewards.

In Bitcoin, selfish mining is a real concern as attackers having any fraction of the whole computational power could successfully use this strategy [20].

Interestingly, our PoI-based Blockchain is less sensible to such attack. Our algorithm gives a protection by design. Indeed, when generating a PoI, a node has to ask to a lots of other nodes to sign messages containing the hash of the previous block, forcing it to reveal any private blocks. Other nodes in the network will request the missing block before accepting to sign the message. In other words, it is not possible to generate a PoI alone. Moreover, if a node is working on a branch that is smaller than the legitimate chain and ask for the signature of an honest node, the latter will tell the former to update its local Blockchain, thus preventing him from wasting resources.

Shared Mining. During the PoI, a node will most of the time be waiting for the signature of another node. So the network delay has the highest impact on the block creation time. To remove this delay, a set of malicious nodes can share theirs private keys between each other and try to create a set S where every nodes are malicious. If one malicious node of the pool succeeds, it can compute the proof locally without sending any messages. It will generate the PoI faster than honest nodes and have a high chance to win.

We defined earlier that each node of the network is known. Which mean that each node is a distinct entity. For this attack to succeed, entities need to share their private keys. This is a very risky move because once you give your private key to someone, he can create transactions in your name without your authorization. This risk alone should discourage honest nodes to do it, even if they want to maximize their gain.

We can still assume that a small number of malicious nodes do know each other and collude to perform this attack. We show now that this attack is hard to perform. S only depends on the previous block and on the identity of the initiator of the proof, so the nodes have no control over it. S consists of n_S nodes randomly selected among the network. So if there are F malicious friends on the network, there is on average the same fraction (n/F) of malicious friends in S as in $\mathcal{N}$. However, the probability for the tour to contains only malicious friend is very low. Indeed, with F malicious friends on the network, the probability that the entire tour consists of malicious friends is $(n/F)^{mean(D)}$ in average.

When a malicious node initiates a PoI for a given message, it can see whether the tour contains an honest node or not, so it might be tempted to change the content (by reordering the transaction or inserting dummy transactions) of its block until the tour contains only his malicious friends. However, even if there is a fraction $(n/F) = 0.1$ of malicious friends in the network (hence in S), and if $mean(D) = 100$, for instance, then the probability that a given tour contains only malicious friends is 10^{-100}. To find a tour with only malicious friends, an initiator would have to try in average 10^{100} different block content, which is not feasible in practice.

6 Conclusion and Possible Extensions

We have presented an new puzzle mechanism that requires negligible work from all the participants. It asks participants to gather sequentially a list of signatures from a subset of the network, forcing them to wait for the response of each visited node. This mechanism can be easily integrated into a Blockchain protocol, replacing the energy inefficient Proof-of-work. The resulting Blockchain protocol is efficient and more secure than the Bitcoin protocol as it is not subject to selfish-mining. Also, it does not have the security issues found in usual PoW replacements such as Proof-of-stack or Proof-of-elapsed time. However, it currently works only in networks where participants are known in advance. The design of our Blockchain protocol makes it easy to propose a possible extension to remove this assumption.

The easiest way to allow anyone to be able to create blocks, is to select as participants the n nodes that locked the highest amount of money. This technique is similar to several existing blockchain based on protocols that work only with known participants (such as Tendermint [15] using an extension of PBFT [6]) or where the nodes producing blocks are reduced for performance reasons (such as EOS [13] where 21 producer nodes are elected by votes from stakeholders). We believe a vote mechanism from stakeholders can elect the set of participants executing our protocol. The main advantage with our solution is that the number of participants can be very high, especially compared to previously mentioned protocols.

References

[1] Poet 1.0 specification. `https://sawtooth.hyperledger.org/docs/core/releases/1.2.4/architecture/poet.html`

[2] Abliz, M., Znati, T.: A guided tour puzzle for denial of service prevention. In: 2009 Annual Computer Security Applications Conference. pp. 279–288. IEEE (2009)

[3] Alsunaidi, S.J., Alhaidari, F.A.: A survey of consensus algorithms for blockchain technology. In: 2019 International Conference on Computer and Information Sciences (ICCIS). pp. 1–6. IEEE (2019)

[4] Back, A., et al.: Hashcash-a denial of service counter-measure (2002)

[5] Bonnet, F., Bramas, Q., Défago, X.: Stateless distributed ledgers. arXiv preprint arXiv:2006.10985 (2020)

[6] Castro, M., Liskov, B.: Practical byzantine fault tolerance and proactive recovery. ACM Transactions on Computer Systems (TOCS) **20**(4), 398–461 (2002)

[7] CBECI: Cambridge bitcoin electricity consumption index (2020), `https://www.cbeci.org`

[8] Chen, L., Xu, L., Shah, N., Gao, Z., Lu, Y., Shi, W.: On security analysis of proof-of-elapsed-time (poet). In: Spirakis, P., Tsigas, P. (eds.) Stabilization, Safety, and Security of Distributed Systems. Springer International Publishing

[9] Douceur, J.R.: The sybil attack. In: International workshop on peer-to-peer systems. pp. 251–260. Springer (2002)

[10] Eyal, I., Sirer, E.G.: Majority is not enough: Bitcoin mining is vulnerable. In: International conference on financial cryptography and data security. pp. 436–454. Springer (2014)

[11] Gallersdörfer, U., Klaaßen, L., Stoll, C.: Energy consumption of cryptocurrencies beyond bitcoin. Joule (2020)

[12] Gaži, P., Kiayias, A., Russell, A.: Stake-bleeding attacks on proof-of-stake blockchains. In: 2018 Crypto Valley Conference on Blockchain Technology (CVCBT). pp. 85–92. IEEE (2018)

[13] IO, E.: Eos. io technical white paper. EOS. IO (accessed 18 December 2017) https://github.com/EOSIO/Documentation (2017)

[14] King, S., Nadal, S.: Ppcoin: Peer-to-peer crypto-currency with proof-of-stake. self-published paper, August **19** (2012)

[15] Kwon, J.: Tendermint: Consensus without mining. Draft v. 0.6, fall **1**(11) (2014)

[16] Mirkovic, J., Dietrich, S., Dittrich, D., Reiher, P.: Internet denial of service: attack and defense mechanisms (Radia Perlman Computer Networking and Security). Prentice Hall PTR (2004)

[17] Mukhopadhyay, U., Skjellum, A., Hambolu, O., Oakley, J., Yu, L., Brooks, R.: A brief survey of cryptocurrency systems. In: 2016 14th annual conference on privacy, security and trust (PST). pp. 745–752. IEEE (2016)

[18] Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system.(2008) (2008)

[19] Preneel, B.: Analysis and design of cryptographic hash functions. Ph.D. thesis, Katholieke Universiteit te Leuven (1993)

[20] Sapirshtein, A., Sompolinsky, Y., Zohar, A.: Optimal selfish mining strategies in bitcoin. In: International Conference on Financial Cryptography and Data Security. pp. 515–532. Springer (2016)

[21] Tritilanunt, S., Boyd, C., Foo, E., González Nieto, J.M.: Toward non-parallelizable client puzzles. In: Bao, F., Ling, S., Okamoto, T., Wang, H., Xing, C. (eds.) Cryptology and Network Security. pp. 247–264. Springer Berlin Heidelberg, Berlin, Heidelberg (2007)

[22] Wang, W., Hoang, D.T., Hu, P., Xiong, Z., Niyato, D., Wang, P., Wen, Y., Kim, D.I.: A survey on consensus mechanisms and mining strategy management in blockchain networks. IEEE Access **7**, 22328–22370 (2019)