A Proof-of-Stake protocol for consensus on Bitcoin subchains

Massimo Bartoletti, Stefano Lande, and Alessandro Sebastian Podda

Università degli Studi di Cagliari, Italy

Abstract. Although the transactions on the Bitcoin blockchain have the main purpose of recording currency transfers, they can also carry a few bytes of metadata. A sequence of transaction metadata forms a *subchain* of the Bitcoin blockchain, and it can be used to store a tamper-proof execution trace of a smart contract. Except for the trivial case of contracts which admit any trace, in general there may exist *inconsistent* subchains which represent incorrect contract executions. A crucial issue is how to make it difficult, for an adversary, to subvert the execution of a contract by making its subchain inconsistent. Existing approaches either postulate that subchains are always consistent, or give weak guarantees about their security (for instance, they are susceptible to Sybil attacks). We propose a consensus protocol, based on Proof-of-Stake, that incentivizes nodes to consistently extend the subchain. We empirically evaluate the security of our protocol, and we show how to exploit it as the basis for smart contracts on Bitcoin.

1 Introduction

Recently, cryptocurrencies like Bitcoin [26] have pushed forward the concept of decentralization, by ensuring reliable interactions among mutually distrusting nodes in the presence of a large number of colluding adversaries. These cryptocurrencies leverage on a public data structure, called *blockchain*, where they permanently store all the transactions exchanged by nodes. Adding new blocks to the blockchain (called *mining*) requires to solve a moderately difficult cryptographic puzzle. The first miner who solves the puzzle earns some virtual currency (some fresh coins for the mined block, and a small fee for each transaction included therein). In Bitcoin, miners must invert a hash function whose complexity is adjusted dynamically in order to make the average time to solve the puzzle $\sim$10 minutes. Instead, removing or modifying existing blocks is computationally unfeasible: roughly, this would require an adversary with more *hashing power* than the rest of all the other nodes. If modifying or removing blocks were computationally easy, an attacker could perform a *double-spending* attack where he pays some amount of coins to a merchant (by publishing a suitable transaction in the blockchain) and then, after he has received the item he has paid for, removes the block containing the transaction. According to the folklore, Bitcoin would resist to attacks unless the adversaries control the majority of total computing power of the Bitcoin network. Even though some vulnerabilities have been reported in

the literature (see Section 4), in practice Bitcoin has worked surprisingly well so far: indeed, the known successful attacks to Bitcoin are standard hacks or frauds [19], unrelated to the Bitcoin protocol.

The idea of using the Bitcoin blockchain and its consensus protocol as foundations for *smart contracts* — namely, decentralized applications beyond digital currency [29] — has been explored by several recent works. For instance, [3,5,7,9,22,23,24] propose protocols for secure multiparty computations and fair lotteries; [13] implements decentralised authorization systems on Bitcoin, [28,30] allow users to log statements on the blockchain; [10] is a key-value database with get/set operations; [14] extends Bitcoin with advanced financial operations (like e.g., creation of virtual assets, payment of dividends, *etc.*), by embedding its own messages in Bitcoin transactions.

Although the Bitcoin blockchain is primarily intended to trade currency, its protocol allows clients to embed a few extra bytes as metadata in transactions. Many platforms for smart contracts exploit these metadata to store a persistent, timestamped and tamper-proof historical record of all their messages [1,6]. Usually, metadata are stored in `OP_RETURN` transactions [2], making them meaningless to the Bitcoin network and unspendable. With this approach, the sequence of platform-dependent messages forms a *subchain*, whose content can only be interpreted by the nodes that execute the platform (we refer to them as *meta-nodes*, to distinguish them from Bitcoin nodes). However, since the platform logic is separated from the Bitcoin logic, a meta-node can append to the subchain transactions with metadata which are meaningless for the platform — or even *inconsistent* with the intended execution of the smart contract. As far as we know, none of the existing platforms use a secure protocol to establish if their subchain is consistent. This is a serious issue, because it either limits the expressiveness of the smart contracts supported by these platforms (which must consider all messages as consistent, so basically losing the notion of state), or degrades the security of contracts (because adversaries can manage to publish inconsistent messages, so tampering with the execution of smart contracts).

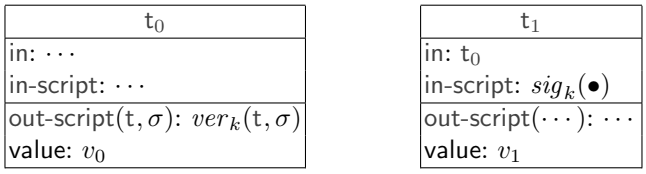
Contributions. We propose a protocol that allows meta-nodes to maintain a consistent subchain over the Bitcoin blockchain. Our protocol is based on *Proof-of-Stake* [8,21], since extending the subchain must be endorsed with a money deposit. Intuitively, a meta-node which publishes a consistent message gets back its deposit once the message is confirmed by the rest of the network. In particular, our protocol provides an economic incentive to honest meta-nodes, while disincentivizing the dishonest ones. We empirically validate the security of our protocol by simulating it in various attack scenarios. Notably, our protocol can be implemented in Bitcoin by only using the so-called *standard* transactions[1].

[1] This is important, because non-standard transactions are discarded by nodes running the official Bitcoin client.

2 Bitcoin and the blockchain

Bitcoin is a cryptocurrency and a digital open-source payment infrastructure that has recently reached a market capitalization of almost $30 billions[2]. The Bitcoin network is peer-to-peer, not controlled by any central authority [26]. Each Bitcoin user owns one or more personal wallets, which consist of pairs of asymmetric cryptographic keys: the public key uniquely identifies the user *address*, while the private key is used to authorize payments. *Transactions* describe transfers of bitcoins (Ƀ), and the history of all transactions, which recorded on a public, immutable and decentralised data structure called *blockchain*, determines how many bitcoins are contained in each address.

To explain how Bitcoin works, we consider two transactions $\mathtt{t}_0$ and $\mathtt{t}_1$, which we graphically represent as follows:[3]



The transaction $\mathtt{t}_0$ contains v_0Ƀ, which can be *redeemed* by putting on the blockchain a transaction (e.g., $\mathtt{t}_1$), whose in field is the cryptographic hash of the whole $\mathtt{t}_0$ (for simplicity, just displayed as $\mathtt{t}_0$ in the figure). To redeem $\mathtt{t}_0$, the in-script of $\mathtt{t}_1$ must contain values making the out-script of $\mathtt{t}_0$ (a boolean programmable function) evaluate to true. When this happens, the value of $\mathtt{t}_0$ is transferred to the new transaction $\mathtt{t}_1$, and $\mathtt{t}_0$ is no longer redeemable. Similarly, a new transaction can then redeem $\mathtt{t}_1$ by satisfying its out-script.

In the example displayed above, the out-script of $\mathtt{t}_0$ evaluates to true when receiving a digital signature σ on the redeeming transaction t, with a given key pair k. We denote with $ver_k(\mathtt{t}, \sigma)$ the signature verification, and with $sig_k(\bullet)$ the signature of the enclosing transaction ($\mathtt{t}_1$ in our example), including *all* the parts of the transaction *except* its in-script.

Now, assume that the blockchain contains $\mathtt{t}_0$, not yet redeemed, when someone tries to append $\mathtt{t}_1$. To validate this operation, the nodes of the Bitcoin network check that $v_1 \leq v_0$, and then they evaluate the out-script of $\mathtt{t}_0$, by instantiating its formal parameters t and σ, to $\mathtt{t}_1$ and to the signature $sig_k(\bullet)$, respectively. The function ver_k verifies that the signature is correct: therefore, the out-script succeeds, and $\mathtt{t}_1$ redeems $\mathtt{t}_0$.

Bitcoin transactions may be more general than the ones illustrated by the previous example: their general form is displayed in Figure 1. First, there can be multiple inputs and outputs (denoted with array notation in the figure). Each output has an associated out-script and value, and can be redeemed independently from others. Consequently, in fields must specify which output they are

[2] Source: crypto-currency market capitalizations http://coinmarketcap.com

[3] in-script and out-script are respectively referred as scriptPubKey and scriptSig in the Bitcoin documentation.

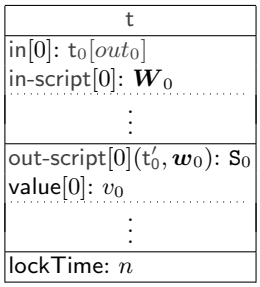


Fig. 1: General form of transactions.

redeeming ($\mathsf{t}_0[out_0]$ in the figure). Similarly, a transaction with multiple inputs associates an in-script to each of them. To be valid, the sum of the values of all the inputs must be greater or equal to the sum of the values of all outputs. In its general form, the out-script is a program in a (not Turing-complete) scripting language, featuring a limited set of logic, arithmetic, and cryptographic operators. Finally, the lockTime field specifies the earliest moment in time (block number or UNIX timestamp) when the transaction can appear on the blockchain.

The Bitcoin network is populated by a large set nodes, called *miners*, which collect transactions from clients, and are in charge of appending the valid ones to the blockchain. To this purpose, each miner keeps a local copy of the blockchain, and a set of unconfirmed transactions received by clients, which it groups into *blocks*. The goal of miners is to add these blocks to the blockchain, in order to get a revenue. Appending a new block B_i to the blockchain requires miners to solve a cryptographic puzzle, which involves the hash $h(B_{i-1})$ of block B_{i-1}, a sequence of unconfirmed transactions $\langle T_i \rangle_i$, and some salt R. More precisely, miners have to find a value of R such $h(h(B_{i-1}) \| \langle T_i \rangle_i \| R) < \mu$, where the value μ is adjusted dynamically, depending on the current hashing power of the network, to ensure that the average mining rate is of 1 block every 10 minutes. The goal of miners is to win the "lottery" for publishing the next block, i.e. to solve the cryptopuzzle before the others; when this happens, the miner receives a reward in newly generated bitcoins, and a small fee for each transaction included in the mined block. If a miner claims the solution of the current cryptopuzzle, the others discard their attempts, update their local copies of the blockchain with the new block B_i, and start mining a new block on top of B_i. In addition, miners are asked to verify the validity of the transactions in B_i by executing the associated scripts. Although verifying transactions is not mandatory, miners are incentivized to do that, because if in any moment a transaction is found invalid, they lose the fee earned when the transaction was published in the blockchain.

If two or more miners solve a cryptopuzzle simultaneously, they create a *fork* in the blockchain (i.e., two or more parallel valid branches). In the presence of a fork, miners must choose a branch wherein carrying out the mining process; roughly, this divergence is resolved once one of the branches becomes longer

than the others. When this happens, the other branches are discarded, and all the orphan transactions contained therein are nullified.

Overall, this protocol essentially implements a *"Proof-of-Work"* system [15].

3 A protocol for consensus on Bitcoin subchains

We define the notions of subchain and consistency in Section 3.1. In Section 3.2 we describe our protocol to embed consistent subchains on the Bitcoin blockchain; we examine some of its properties in Section 3.3. Finally, in Section 3.4 we show how to implement our protocol in Bitcoin.

3.1 Subchains and consistency

We assume a set $\mathsf{A}, \mathsf{B}, \ldots$ of participants, who want to append messages $\mathtt{a}, \mathtt{b}, \ldots$ to the subchain. A *label* is a pair containing a participant A and a message $\mathtt{a}$, written $\mathsf{A} : \mathtt{a}$. *Subchains* are finite sequences of labels, written $\mathsf{A}_1 : \mathtt{a}_1 \cdots \mathsf{A}_n : \mathtt{a}_n$, which are embedded in the Bitcoin blockchain. The intuition is that A_1 has embedded the message $\mathtt{a}_1$ in some transaction $\mathtt{t}_1$ of the Bitcoin blockchain, then A_2 has appended some transaction $\mathtt{t}_2$ embedding $\mathtt{a}_2$, and so on. For a subchain η, we write $\eta\ \mathsf{A} : \mathtt{a}$ for the subchain obtained by appending $\mathsf{A} : \mathtt{a}$ to η.

In general, labels can also have side effects on the Bitcoin blockchain: we represent with $\mathsf{A} : \mathtt{a}(v \to \mathsf{B})$ a label which also transfers v฿ from A to B. When this message is on the subchain, it also acts as a standard currency transfer on the Bitcoin blockchain, which makes v฿ in a transaction of A redeemable by B. When the value v is zero or immaterial, we simply write $\mathtt{a}$ instead of $\mathtt{a}(v \to \mathsf{B})$.

A crucial insight is that not all possible sequences of labels are valid subchains: to define the *consistent* ones, we interpret subchains as traces of *Labelled Transition Systems* (LTS). Formally, an LTS is a tuple $(Q, L, q_0, \to)$, where:

- Q is a set of states (ranged over by $q, q', \ldots$);
- L is a set of labels (in our case, of the form $\mathsf{A} : \mathtt{a}$);
- $q_0 \in Q$ is the initial state;
- $\to \subseteq Q \times L \times Q$ is a transition relation.

As usual, we write $q \xrightarrow{\mathsf{A}:\mathtt{a}} q'$ when $(q, \mathsf{A} : \mathtt{a}, q') \in \to$, and, given a subchain $\eta = \mathsf{A}_1 : \mathtt{a}_1 \cdots \mathsf{A}_n : \mathtt{a}_n$, we write $q \xrightarrow{\eta} q'$ whenever there exist $q_1, \ldots, q_n$ such that:

$$q \xrightarrow{\mathsf{A}_1:\mathtt{a}_1} q_1 \xrightarrow{\mathsf{A}_2:\mathtt{a}_2} \cdots \xrightarrow{\mathsf{A}_n:\mathtt{a}_n} q_n = q'$$

We require that the relation $\to$ is *deterministic*, i.e. if $q \xrightarrow{\mathsf{A}:\mathtt{a}} q'$ and $q \xrightarrow{\mathsf{A}:\mathtt{a}} q''$, then it must be $q' = q''$.

The intuition is that the subchain has a state (initially, q_0), and each message updates the state according to the transition relation. More precisely, if the subchain is in state q, then a message $\mathtt{a}$ sent by A makes the state evolve to q' whenever $q \xrightarrow{\mathsf{A}:\mathtt{a}} q'$ is a transition in the LTS.

Note that, for some state q and label $\mathsf{A}:\mathtt{a}$, it may happen that no state q' exists such that $q \xrightarrow{\mathsf{A}:\mathtt{a}} q'$. In this case, if q is the current state of the subchain, we want to make hard for a participant (possibly, an adversary trying to tamper with the subchain) to append such message. Informally, a subchain $\mathsf{A}_1:\mathtt{a}_1 \cdots \mathsf{A}_n:\mathtt{a}_n$ is *consistent* if, starting from the initial state q_0, it is possible to find states $q_1, \ldots, q_n$ such that from each q_i there is a transition labelled $\mathsf{A}_{i+1}:\mathtt{a}_{i+1}$ to q_{i+1}.

Definition 1 (Subchain consistency). *We say that a subchain η is* consistent *whenever there exists q such that $q_0 \xrightarrow{\eta} q$.*

Note that, if a subchain is consistent, then by determinism we have that the state q_n exists and is unique. In other words, a consistent sequence of messages uniquely identifies the state of the subchain.

Example 1. To illustrate consistency, consider a smart contract $\mathtt{FACTORS}_n$ which rewards with 1Ƀ each participant who extends the subchain with a new prime factor of n. The contract accepts two kinds of messages:

- $\mathtt{send}_p$, where p is a natural number;
- $\mathtt{pay}_p(1 \to \mathsf{A})$, meaning that A receives a reward for the factor p;

The states of the contract can be represented as sets of triples (A, p, b), where b is a boolean value indicating whether A has been rewarded for the factor p. The initial state is $\emptyset$. We define the transition relation of $\mathtt{FACTORS}_n$ as follows:

- $S \xrightarrow{\mathsf{A}:\mathtt{send}_p} S'$, iff p is a prime factor of n, $(\mathsf{B}, p, b) \notin S$ for any B and b, and $S' = S \cup \{(\mathsf{A}, p, 0)\}$;
- $S \xrightarrow{\mathsf{F}:\mathtt{pay}_p(1 \to \mathsf{A})} S'$, iff $(\mathsf{A}, p, 0) \in S$ and $S' = (S \setminus \{(\mathsf{A}, p, 0)\}) \cup \{(\mathsf{A}, p, 1)\}$.

Consider now the following subchains for $\mathtt{FACTORS}_{330}$, where F is the participant who issues the contract, and M is an adversary:

1. $\eta_1 = \mathsf{A}:\mathtt{send}_{11}\ \ \mathsf{B}:\mathtt{send}_2\ \ \mathsf{F}:\mathtt{pay}_{11}(1 \to \mathsf{A})\ \ \mathsf{F}:\mathtt{pay}_2(1 \to \mathsf{B})$
2. $\eta_2 = \mathsf{A}:\mathtt{send}_{11}\ \ \mathsf{F}:\mathtt{pay}_{11}(1 \to \mathsf{A})\ \ \mathsf{M}:\mathtt{send}_{11}$
3. $\eta_3 = \mathsf{M}:\mathtt{send}_{229}\ \mathsf{F}:\mathtt{pay}_{229}(1 \to \mathsf{M})$
4. $\eta_4 = \mathsf{A}:\mathtt{send}_{11}\ \ \mathsf{F}:\mathtt{pay}_{11}(1 \to \mathsf{M})$

The subchain η_1 is consistent, because both A and B send new factors and get their rewards. The subchains η_2 and η_3 are inconsistent, because 11 sent by M is not fresh, and 229 is not a factor of 330. Finally, the subchain η_4 is inconsistent, because M gets the reward that should have gone to A. □

Similarly to Bitcoin, we do not aim at guaranteeing that a subchain is *always* consistent. Indeed, also in Bitcoin a miner could manage to append a block with invalid transactions: in this case, as discussed in Section 2, the Bitcoin blockchain forks, and the other miners must choose which branch to follow. However, honest miners will neglect the branch with invalid transactions, so eventually (since honest miners detain the majority of computational power), that branch will be abandoned by all miners.

For subchain consistency we adopt a similar notion: we assume that an adversary can append a label $\mathsf{A}:\mathtt{a}$ such that $q_n \not\xrightarrow{\mathsf{A}:\mathtt{a}}$, so making the subchain inconsistent. However, upon receiving such label, honest nodes will discard it. To formalise their behaviour, we define below a function Γ that, given a subchain η (possibly inconsistent), filters all the invalid messages. Hence, $\Gamma(\eta)$ is a consistent subchain.

Definition 2 (Branch pruning). *We inductively define the endofunction Γ on subchains as follows, where ϵ denotes the empty subchain:*

$$\Gamma(\epsilon) = \epsilon \qquad \Gamma(\eta\ \mathsf{A}:\mathtt{a}) = \begin{cases} \Gamma(\eta)\ \mathsf{A}:\mathtt{a} & \text{if } \exists q, q' : q_0 \xrightarrow{\Gamma(\eta)} q \xrightarrow{\mathsf{A}:\mathtt{a}} q' \\ \Gamma(\eta) & \text{otherwise} \end{cases}$$

In order to model which labels can be appended to the subchain without breaking its consistency, we introduce below the auxiliary relation $\models$. Informally, given a consistent subchain η, the relation $\eta \models \mathsf{A}:\mathtt{a}$ holds whenever the subchain $\eta\,\mathsf{A}:\mathtt{a}$ is still consistent.

Definition 3 (Consistent update). *We say that $\mathsf{A}:\mathtt{a}$ is a* consistent update *of a subchain η, denoted with $\eta \models \mathsf{A}:\mathtt{a}$, iff the subchain $\Gamma(\eta)\ \mathsf{A}:\mathtt{a}$ is consistent.*

Example 2. Recall the subchain $\eta_2 = \mathsf{A}:\mathtt{send}_{11}\ \mathsf{F}:\mathtt{pay}_{11}(1 \to \mathsf{A})\ \mathsf{M}:\mathtt{send}_{11}$ from Example 1. We have that $\mathsf{B}:\mathtt{send}_2$ is a consistent update of η_2, because $\Gamma(\eta_2)\ \mathsf{B}:\mathtt{send}_2 = \mathsf{A}:\mathtt{send}_{11}\ \mathsf{F}:\mathtt{pay}_{11}(1 \to \mathsf{A})\ \mathsf{B}:\mathtt{send}_2$ is consistent. □

3.2 Description of the protocol

Assume a network of mutually distrusted nodes $\mathsf{N}, \mathsf{N}', \ldots$, that we call *meta-nodes* to distinguish them from the nodes of the Bitcoin network. Meta-nodes receive messages from participants (also mutually distrusting) which want to extend the subchain. Our goal is to allow honest participants (i.e., those who follow the protocol) to perform consistent updates of the subchain, while disincentivizing adversaries who attempt to make the subchain inconsistent.

To this purpose, we propose a protocol based on *Proof-of-Stake* (PoS). Namely, we rely on the assumption that the overall stake retained by honest participants is greater than the stake of dishonest ones[4]. The stake is needed by meta-nodes, which have to vote for approving messages sent by participants. These messages are embedded into Bitcoin transactions, which we call *update requests*. We denote by $\mathsf{UR}[\mathsf{A}:\mathtt{a}]$ the update request issued by A to append the message $\mathtt{a}$ to the subchain. In order to vote an update request, a meta-node must invest κ฿ on it, where κ is a constant specified by the protocol. An update request needs the vote of a single meta-node. The protocol requires meta-nodes to vote a request $\mathsf{UR}[\mathsf{A}:\mathtt{a}]$ only if $\mathsf{A}:\mathtt{a}$ is a consistent update of the current subchain η, i.e. if

[4] Note that a similar hypothesis, but related to computational power rather than stake, holds in Bitcoin, where honest miners are supposed to control more computational power than dishonest ones.

1. Upon receiving an update request $\mathsf{UR}[\mathsf{A}:\mathtt{a}]$, a meta-node checks its consistency, $\eta \models \mathsf{A}:\mathtt{a}$. If so, it votes the request, and adds it to the request pool;
2. when Δ expires, the arbiter signs all the well-formed UR in the request pool;
3. all requests signed by the arbiter are sent to the Bitcoin miners, to be published on the blockchain. The first to be mined, indicated with UR_i, is the i-th label of the subchain.

Fig. 2: Summary of a protocol stage i.

$\eta \models \mathsf{A}:\mathtt{a}$[5]. To incentivize meta-nodes to vote their update requests, participants pay them a *fee* (smaller than κ), which can be redeemed by meta-nodes when the update request is appended to the subchain.

We define our protocol in Figure 2. It is organised in *stages*. The protocol ensures that *exactly one* label $\mathsf{A}:\mathtt{a}$ is appended to the subchain for each stage i. This is implemented by appending a corresponding transaction $\mathsf{UR}_i[\mathsf{A}:\mathtt{a}]$ to the Bitcoin blockchain. To guarantee its uniqueness, the protocol exploits an *arbiter* T, namely a distinguished node of the network which is assumed honest (we discuss this hypothesis in Section 3.3). We now describe the main steps of the protocol.

At step 1 of the stage i of the protocol, a meta-node (say, N) votes an update request (as detailed in Section 3.4). In order to do this, N must confirm a previous update UR_j in the subchain, by paying κ ฿ (plus the participant's fee) to the meta-node N' who appended UR_j to the subchain. To avoid the *self-compensation attack* discussed later on in Section 3.3, the protocol only allows to confirm one of the past C updates, where $C \geq 2$ is a constant fixed by the protocol (called *checkpoint offset*). Summing up, the value j is such that: (i) $j < i$; (ii) $|i - j| < C$; (iii) $\mathsf{UR}_j[\mathsf{A}:\mathtt{a}]$ is consistent. In this way the protocol incentivizes meta-nodes to vote consistent updates only, since inconsistent ones are not likely to be confirmed. If all the last C updates in the subchain are inconsistent, then N chooses the last one. Then, N adds $\mathsf{UR}[\mathsf{A}:\mathtt{a}]$ to the *request pool*, i.e. the set of all voted requests of the current stage (emptied at the beginning of each stage). This voting step has a fixed duration Δ, specified by the protocol (the choice of Δ is discussed in Section 5).

At step 2, which starts when Δ expires, the arbiter T signs all *well-formed* request transactions, i.e., those respecting the format defined in Section 3.4.

At step 3, meta-nodes send the requests signed by T to the Bitcoin network. The mechanism described in Section 3.4 ensures that, at each stage i, exactly one transaction, denoted $\mathsf{UR}_i[\mathsf{A}:\mathtt{a}]$, is put on the Bitcoin blockchain. When this happens, the label $\mathsf{A}:\mathtt{a}$ is appended to the subchain.

[5] We assume that all meta-nodes agree on the Bitcoin blockchain; since η is a projection of the blockchain, they also agree on η.

3.3 Basic properties of the protocol

We now establish some basic properties of our protocol. Hereafter, we assume that honest nodes control the majority of the total stake of the network[6], hereafter denoted by S. Further, we assume that the overall stake required to vote pending update requests is greater than the overall stake of honest meta-nodes.

Adversary power. An honest meta-node votes as many requests as is allowed by its stake. Hence, if its stake is h, it votes h/κ requests per stage. Consequently, the rest of the network — which may include dishonest meta-nodes not following the protocol — can vote at most $(S-h)/\kappa$ requests. Then:

Proposition 1. *The probability that an honest meta-node with stake h updates the subchain is at least h/S at each stage.*

Since we assume that honest meta-nodes control the majority of the stake, Proposition 1 also limits the capabilities of the adversary:

Proposition 2. *If the global stake of honest meta-nodes is S_H, then dishonest ones update the subchain with probability at most $(S-S_H)/S$ at each stage.*

Although inconsistent updates are ignored by honest meta-nodes, their side effects as standard Bitcoin transactions (i.e. trasfers of v฿ from A to B in labels $\mathsf{A} : \mathsf{a}(v \to \mathsf{B})$) cannot be revoked once they are included in the Bitcoin blockchain. We now show how the incentive system in our protocol reduces the feasibility of such inconsistent updates.

Assume that an adversary M manages to append 2 updates to the subchain: an inconsistent update at index j, and a consistent one at index $i > j$. Since M does not follow the protocol, she can exploit UR_i to redeem the κ฿ she put on UR_j. Later on, the adversary will be able to redeem the κ฿ she put on UR_i: indeed, honest meta-nodes will vote UR_i, as it is consistent. We call the above behaviour of M *self-compensation attack*.

Now, according to Proposition 2, if M has stake m, and the other meta-nodes are honest, then M has probability at most m/S of extending the subchain in a given stage of the protocol. Since stages can be seen as independent events, and since M has to publish at least 2 updates over the most recent checkpoint to perform the attack, we obtain the following:

Proposition 3. *The probability that an adversary with stake m succeeds in a self-compensation attack is at most:*

$$\binom{C}{2} \cdot \mu^2 (1-\mu)^{C-2}$$

where C is the checkpoint offset, and $\mu = m/S$.

[6] Under this assumption, meta-nodes can ensure that the arbiter is honest.

Since the probability to publish inconsistent updates without losing κ฿ grows with C, it is crucial to keep this value small. For instance, if $\mu = 0.1$ an adversary could perform the attack with probability bounded by (i) 0.01 if $C = 2$; (ii) 0.027 if $C = 3$; (iii) 0.0486 if $C = 4$.

Observe that if the attack succeeds once, then the attack probability slightly increases, since the stake m is charged by the client fees of the published updates. This is not an issue if the fee is small compared to S.

Trustworthiness of the arbiter. Our protocol uses in arbiter T to ensures that exactly one transaction per stage is appended to the blockchain, as well its choice is random. In order to simplify the description of the protocol, we have assumed the arbiter T to behave honestly. However, our arbiter does not play the role of a trusted authority: indeed, the update requests to be voted are chosen by the meta-nodes, and once they are added to the request pool, the arbiter is expected to sign all of them, without taking part on the validation nor in the voting. Since everyone can inspect the request pool, any misbehaviour of the arbiter can be detected by the meta-nodes, which can proceed to replace it.

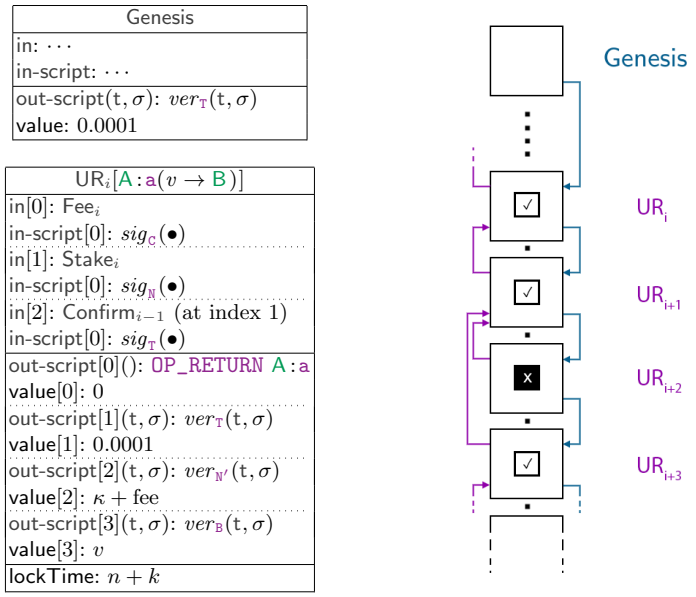
3.4 Implementation in Bitcoin

In this section we show how our protocol can be implemented in Bitcoin. A label $\mathsf{A}:\mathtt{a}(v \to \mathsf{B})$ at position i of the subchain is implemented as the Bitcoin transaction $\mathsf{UR}_i[\mathsf{A}:\mathtt{a}(v \to \mathsf{B})]$ in Figure 3a, with the following outputs:

- the output of index 0 embeds the label $\mathsf{A}:\mathtt{a}$. This is implemented through an unspendable `OP_RETURN` script [6][7].
- the output of index 1 links the transaction to the previous element of the subchain, pointed by $\mathsf{in}[2]$. This link requires the arbiter signature. Note that, since all the update requests in the same stage redeem the same output, exactly one of them can be mined.
- the output of index 2 implements the incentive mechanism. The script rewards the meta-node N' which has voted a preceding UR_j in the subchain. Meta-node N' can redeem from this output κ฿ plus the participant's fee, by providing his signature.
- the output of index 3 is only relevant for messages $\mathtt{a}(v \to \mathsf{B})$ where $v > 0$. Participant B can redeem v฿ from this output by providing his signature.

All transactions specify a $\mathsf{lockTime}$ $n + k$, where n is the current Bitcoin block number, and k is a positive constant. This ensures that a transaction can be mined only after k blocks. In this way, even if a transaction is signed by the arbiter and sent to miners before the others, it has the same probability as the others of being appended to the blockchain.

[7] The `OP_RETURN` instruction allows to save 80 bytes metadata in a transaction; an $\mathsf{out\text{-}script}$ containing `OP_RETURN` always evaluates to false, hence it is unspendable.



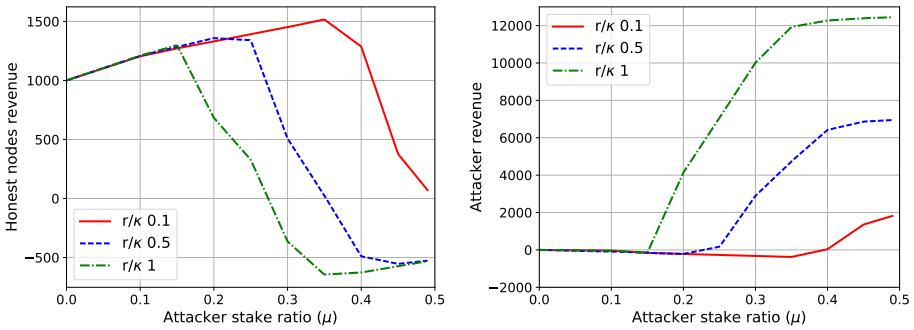
(a) (b)

Fig. 3: In (a), format of Bitcoin transactions used to implement our protocol. In (b), a subchain mantained through our protocol. Since UR_{i+2} contains an inconsistent update, the meta-node which voted it is not rewarded.

To initialise the subchain, the arbiter puts the Genesis transaction on the Bitcoin blockchain. This transaction secures a small fraction of bitcoin, which can be redeemed by UR_1 through the arbiter signature. This value is then transferred to each subsequent update of the subchain (see Figure 3b). At each protocol stage, participants send incomplete UR transactions to the network. These transactions contain only in[0] and out[0], specifying the fee and the message for the subchain (including the value to be transferred). To vote, meta-nodes add in[1], in[2] and out[2] to these transactions, to, respectively, put the required κ (from some transaction Stake_i), declare they want extend the last published update $\mathsf{Confirm}_{i-1}$, and specify the previous update to be rewarded. All the in[1] fields in a stage of the protocol must be different, to prevent attackers to vote more URs with the same funds.

4 Evaluation of the protocol

In this section we evaluate the security of our protocol, providing some experimental results. We also investigate how possible attacks to Bitcoin may affect subchains built on top of its blockchain.



(a) (b)

Fig. 4: Revenue of honest nodes (a) and of the attacker M (b) for increasing values of the attacker stake ratio μ. The curves represent different values of r/κ (the ratio between the attack revenue r, given by inconsistent $\mathsf{a}(r \to \mathsf{M})$ updates, and the cost of the vote).

Attack scenario. We assume an adversary who can craft any update (consistent or not), and controls one meta-node M with stake μS, where $\mu \in [0;1]$ and S is the total stake of the network[8]. We suppose that each meta-node can vote as many update requests as possible, spending all its stake, and that the network is always saturated with pending updates, which globally amount to the entire stake of honest meta-nodes[9]. We also assume that M gets an additional extra revenue r for each inconsistent update, modelling the case where she manages to induce a victim to publish an inconsistent payment $\mathsf{a}(r \to \mathsf{M})$. The goal of M is to append at least 2 updates to the blockchain (one of which inconsistent) every C published updates. She can use any possible strategy to achieve this goal.

We simulate the protocol under the attack scenario described above. Each simulation runs the protocol to generate a subchain with $10,000$ messages, setting the client fee to 0.1κ and the checkpoint offset to 3. To this purpose we use Desmo-J [18], a discrete event simulator for Java.

Experimental results. Figure 4b measures the attacker revenue as μ increases. In particular, it shows that if the stake threshold κ is ten times greater than r, M gains only if she owns at least $\sim$40% of the global stake (i.e., $\mu \geq 0.4$). Therefore, under such assumption about the attacker stake, the security of our protocol is

[8] Assuming a single adversary is not less general than having many non-colluding meta-nodes which carry on individual attacks. Indeed, in this setting meta-nodes do not join their funds to increase the stake ratio μ.

[9] Note that saying the update queue is not always saturated is equivalent to model an adversary with a stronger μ: this because honest meta-nodes cannot spend all their stake in a single protocol stage, i.e. reducing their actual *power*. Thus, studying this particular case will not give any additional contribution to the analysis.

comparable with that of the Bitcoin *Proof-of-Work* protocol [17]. Instead, if $\kappa = r$, the attacker needs only ∼15% of the global stake to profit from the attack. Figure 4a shows that, in the absence of attackers ($\mu = 0$), the revenue of honest nodes is essentially the client fee times the number of updates published, as expected. Further, μ is below the threshold required to perform a profitable attack, the revenue of honest nodes increases: this happens because inconsistent updates voted by M reward honest ones, whereas the opposite cannot occur. Summing up, our protocol is secure only if, for updates on the form $\mathtt{a}(r \to \mathsf{A})$, we have that $r \leq \kappa$. Hence, if r is close to 0, the behaving dishonestly is not economically advantageous.

Security of the underlying Bitcoin blockchain. So far we have only considered direct attacks to our protocol, assuming the underlying Bitcoin blockchain to be secure. However, although Bitcoin has been secure in practice till now, some works have spotted some potential vulnerabilities of its protocol. These vulnerabilities could be exploited to execute *Sybil attacks* [4] and *selfish-mining attacks* [16], which might also affect subchains built on top of the Bitcoin blockchain.

In Sybil attacks on Bitcoin, honest nodes are induced to believe that the network is populated by many distinct participants, which instead are controlled by a single malicious entity. This attack is usually exploited to quickly propagate malicious information on the network, and to disguise honest participants in a consensus/reputation protocol, e.g. by overwhelming the network with votes of the adversary. In the selfish-mining attack [16], small groups of colluding miners manage to obtain a revenue larger than the one of honest miners. More specifically, when a selfish-mining pool finds a new block, it keeps it hidden to the rest of the network. In this way, selfish miners gain an advantage over honest ones in mining the next block. This is equivalent to keep a private fork of the blockchain, which is only known to the selfish-mining pool. Note that honest miners still mine on the public branch of the blockchain, and their hash rate is greater than selfish miners' one. Since, in the presence of a fork, the Bitcoin protocol requires to keep mining on the longest chain, selfish miners reveal their private fork to the network just before being overcome by the honest miners. Eyal and Sirer in [16] show that, under certain assumptions, this strategy gives better revenues than honest mining: in the worst scenario (for the adversary), the attack succeeds if the selfish-mining pool controls at least 1/3 of the total hashing power. Rational miners are thus incentivized to join the selfish-mining pool. Once the pool manages to control the majority of the hashing power, the system loses its decentralized nature. Garay, Kiayias and Leonardos in [17] essentially confirm these results: considering a core Bitcoin protocol, they prove that if the hashing power γ of honest miners exceeds the hashing power β of the adversary pool by a factor λ, then the ratio of adversary blocks in the blockchain is bounded by $1/\lambda$ (which is strictly greater than β). Thus, as β (the adversary pool size) approaches 1/2, they control the blockchain.

Although these attacks are mainly related to Bitcoin revenues, they can affect the consistency of any subchain built on top of its blockchain. In particular, suitably adapted versions of these attacks allow adversaries to cheat meta-nodes

about the current subchain state, forcing them to synchronize their local copy of the Bitcoin blockchain with invalid forks that will be discarded by the network in the future. To protect against such attacks, meta-nodes should consider only *l-confirmed* transactions. Namely, if the last published blockchain block is B_n, they consider only those transactions appearing in blocks B_j with $j \leq n - l$. This means that an attacker would have to mine at least l blocks to force the revocation of a *l-confirmed* transaction. Rosenfeld [27] shows that, if an attacker controls at most the 10% of the network hashing power, $l = 6$ is sufficient for reducing the risk of revoking a transaction to less than 0.1%.

5 Discussion

We have proposed a protocol to reach consensus on subchains, i.e. chains of platform-dependent messages embedded in the Bitcoin blockchain. Our protocol incentivizes nodes to validate messages before appending them to the subchain, making economically disadvantageous for an adversary to append inconsistent messages. To confirm this intuition we have measured the security of our protocol over different attack scenarios. Our simulations show that, under conservative assumptions, its security is comparable to that of Bitcoin.

Performance of the protocol. As seen in Section 3.2, the protocol runs in periods of duration Δ. Due to the mechanism for choosing the message to append to the subchain from the request pool, the protocol can publish at most one transaction per Bitcoin block. This means that a lower bound for Δ is the Bitcoin block interval (∼10mins). To monitor the arbiter behaviour throughout protocol stages, all meta-nodes must share a coherent view of the request pool. Then, Δ needs to be large enough to let each node synchronize the request pool with the rest of the network. A possible approach to cope with this issue is to make meta-nodes broadcast their voted updates, and to keep a list of other ones (considering only those which satisfy the format of transactions, as in Section 3.4). More efficient approaches could exploit distributed shared memories [12,20].

Overcoming the metadata size limit. As noted in Section 3.4, we use `OP_RETURN` unspendable scripts to embed metadata in Bitcoin transactions. Since Bitcoin limits the size of such metadata to 80 bytes, this might not be enough to store the data needed by platforms. To overcome this issue, one can use distributed hash tables [25] maintained by meta-nodes. In this way, instead of storing full message data in the blockchain, `OP_RETURN` scripts would contain only the corresponding message digests. The unique identifier of the Bitcoin transaction can be used as the key to retrieve the full message data from the hash table.

Smart contracts over subchains. The model of subchains defined in Section 3.1, based on LTSs, can be easily extended to model the computations of smart contracts over the Bitcoin blockchains. A platform for smart contracts could exploit our model to represent the state of a contract as the state of the subchain, and model its possible state updates through the transition relation.

Implementing a platform for smart contracts would require a language for expressing them. To bridge this language with our abstract model, one can provide the language with an operational semantics, giving rise to an LTS describing the computations. Note that our assumption to model computations as a single LTS does not reduce the generality of the system, since a set of LTSs, each one modelling a contract, can be encoded in one LTS as their parallel composition. If the language is Turing-complete, an additional problem we would have to face is the potential non-termination. This issue has been dealt with in different ways by different platforms. E.g., the approach followed by Ethereum [11] is to impose a fee for each instruction executed by its virtual machine. If the fee does not cover the cost of the whole computation, the execution terminates.

A usable platform must also allow to create new contracts at run-time. Since in our model the LTS representing possible computations is fixed, we would need a mechanism to "extend" it. To handle the publication of new contracts, we could modify the protocol so that UR may contain its code, and the unique identifier of the transaction also identifies the contract. In this extended model, update requests would also contain the identifier of the contract to be updated, so that meta-nodes can execute the corresponding code.

Acknowledgments. This work is partially supported by Aut. Reg. of Sardinia grant P.I.A. 2013 "NOMAD". Alessandro Sebastian Podda gratefully acknowledges Sardinia Regional Government for the financial support of her PhD scholarship (P.O.R. Sardegna F.S.E. Operational Programme of the Autonomous Region of Sardinia, European Social Fund 2007-2013 - Axis IV Human Resources, Objective l.3, Line of Activity l.3.1).

References

1. Making sense of blockchain smart contracts. http://www.coindesk.com/making-sense-smart-contracts/. Last accessed 2017/01/14.
2. opreturn.org. http://opreturn.org/. Last accessed 2016/12/15.
3. M. Andrychowicz, S. Dziembowski, D. Malinowski, and L. Mazurek. Fair two-party computations via Bitcoin deposits. In *Financial Cryptography Workshops*, pages 105–121, 2014.
4. M. Babaioff, S. Dobzinski, S. Oren, and A. Zohar. On Bitcoin and red balloons. In *ACM Conference on Electronic Commerce (EC)*, pages 56–73, 2012.
5. W. Banasik, S. Dziembowski, and D. Malinowski. Efficient zero-knowledge contingent payments in cryptocurrencies without scripts. In *ESORICS*, volume 9879 of *LNCS*, pages 261–280. Springer, 2016.
6. M. Bartoletti and L. Pompianu. An analysis of Bitcoin OP_RETURN metadata. In *Financial Cryptography Workshops*, 2017. Also available as CoRR abs/1702.01024.
7. M. Bartoletti and R. Zunino. Constant-deposit multiparty lotteries on Bitcoin. In *Financial Cryptography Workshops*, 2017. Also available as IACR Cryptology ePrint Archive 955/2016.
8. I. Bentov, A. Gabizon, and A. Mizrahi. Cryptocurrencies without proof of work. In *Financial Cryptography Workshops*, volume 9604 of *LNCS*, pages 142–157. Springer, 2016.

9. I. Bentov and R. Kumaresan. How to use Bitcoin to design fair protocols. In *CRYPTO*, volume 8617 of *LNCS*, pages 421–439. Springer, 2014.
10. Blockstore: Key-value store for name registration and data storage on the Bitcoin blockchain. https://github.com/blockstack/blockstore, 2014.
11. V. Buterin. Ethereum: a next generation smart contract and decentralized application platform. https://github.com/ethereum/wiki/wiki/White-Paper, 2013.
12. M. Cai, A. Chervenak, and M. Frank. A peer-to-peer replica location service based on a distributed hash table. In *ACM/IEEE Conference on High Performance Networking and Computing*, page 56. IEEE Computer Society, 2004.
13. K. Crary and M. J. Sullivan. Peer-to-peer affine commitment using Bitcoin. In *ACM PLDI*, pages 479–488, 2015.
14. R. Dermody, A. Krellenstein, O. Slama, and E. Wagner. CounterParty: Protocol specification. http://counterparty.io/docs/protocol_specification/, 2014.
15. C. Dwork and M. Naor. Pricing via processing or combatting junk mail. In *CRYPTO*, volume 740 of *LNCS*, pages 139–147. Springer, 1993.
16. I. Eyal and E. G. Sirer. Majority is not enough: Bitcoin mining is vulnerable. In *Financial Cryptography and Data Security*, volume 8437 of *LNCS*, pages 436–454. Springer, 2014.
17. J. A. Garay, A. Kiayias, and N. Leonardos. The Bitcoin backbone protocol: Analysis and applications. In *EUROCRYPT*, volume 9057 of *LNCS*, pages 281–310. Springer, 2015.
18. J. Göbel, P. Joschko, A. Koors, and B. Page. The discrete event simulation framework DESMO-J: review, comparison to other frameworks and latest development. In *European Conference on Modelling and Simulation (ECMS)*, pages 100–109. European Council for Modeling and Simulation, 2013.
19. A. Hern. A history of Bitcoin hacks. http://www.theguardian.com/technology/2014/mar/18/history-of-bitcoin-hacks-alternative-currency, march 2014.
20. S. Iyer, A. Rowstron, and P. Druschel. Squirrel: A decentralized peer-to-peer web cache. In *PODC*, pages 213–222. ACM, 2002.
21. A. Kiayias, I. Konstantinou, A. Russell, B. David, and R. Oliynykov. Ouroboros: A provably secure Proof-of-Stake blockchain protocol. *IACR Cryptology ePrint Archive*, 2016:889, 2016.
22. A. Kiayias, H. Zhou, and V. Zikas. Fair and robust multi-party computation using a global transaction ledger. In *EUROCRYPT*, pages 705–734, 2016.
23. R. Kumaresan and I. Bentov. How to use Bitcoin to incentivize correct computations. In *ACM CCS*, pages 30–41, 2014.
24. R. Kumaresan, T. Moran, and I. Bentov. How to use Bitcoin to play decentralized poker. In *ACM CCS*, pages 195–206, 2015.
25. P. Maymounkov and D. Mazières. Kademlia: A peer-to-peer information system based on the XOR metric. In *Workshop on Peer-to-Peer Systems (IPTPS)*, volume 2429 of *LNCS*, pages 53–65. Springer, 2002.
26. S. Nakamoto. Bitcoin: a peer-to-peer electronic cash system. https://bitcoin.org/bitcoin.pdf, 2008.
27. M. Rosenfeld. Analysis of hashrate-based double spending. *CoRR*, abs/1402.2009, 2014.
28. T. Ruffing, A. Kate, and D. Schröder. Liar, liar, coins on fire!: Penalizing equivocation by loss of Bitcoins. In *ACM CCS*, pages 219–230, 2015.
29. N. Szabo. Formalizing and securing relationships on public networks. *First Monday*, 2(9), 1997.
30. A. Tomescu and S. Devadas. Catena: Efficient non-equivocation via Bitcoin. In *IEEE Symp. on Security and Privacy*, 2017.