Soft Power: Upgrading Chain Macroeconomic Policy Through Soft Forks

Anonymous Author

Abstract. Macroeconomic policy in a blockchain system concerns the algorithm that decides the payment schedule for miners and thus its money mint rate. It governs the amounts, distributions, beneficiaries and conditions required for money supply payments to participants by the system. While most chains today employ simple policies such as a constant amount per block, several cryptocurrencies have sprung up that put forth more interesting policies. As blockchains become a more popular form of money, these policies inevitably are becoming more complex. A chain with a simple policy will often need to switch over to a different policy. Until now, it was believed that such upgrades require a hard fork – after all, they are changing the money supply, a central part of the system, and unupgraded miners cannot validate blocks that deviate from those hard-coded rules. In this paper, we present a mechanism that allows a chain to upgrade from one policy to another through a soft fork. Our proposed mechanism works in today's Ethereum blockchain without any changes and can support a very generic class of monetary policies that satisfy a few basic bounds. Our construction is presented in the form of a smart contract. We showcase the usefulness of our proposal by describing several interesting applications of policy changes. Notably, we put forth a mechanism that makes Non-Interactive Proofs of Proof-of-Work unbribable, a previously open problem.

1 Introduction

At the heart of every blockchain [1] system lives a mechanism that distributes rewards to its validators. The mechanism incentivizes miners to mine in proof-of-work [2] chains and minters to mint in proof-of-stake [3] systems. Additionally, it is an ingenius mechanism to distribute new money when no central bank is present.

Today's blockchain systems employ various policies detailing how exactly the proceeds from mining are distributed to miners. Most of these policies are quite simple. For example, Bitcoin's policy rewards the miner of each block with a *constant* amount of bitcoin, currently 12.5 BTC. This amount is halved every four years. Ethereum miners receive 2 ETH, but the system also rewards uncle blocks [4, 5]. A more interesting system is Monero's [6], where a system of *smooth emission* is employed. Instead of *halving* the money supply in a stair function fashion, they slowly decrease the supply block-by-block.

Regardless of what economic policy a chain employs, sooner or later the policy might need to be updated. This becomes necessary as cryptocurrencies are adopted more widely and the community learns about what works better economically. Ethereum's Constantinople hard fork, in which rewards were adjusted [7], constitutes one such example. In fact, as we will showcase in the applications section, some macroeconomic mechanism updates help with incentivizing correct execution of the core consensus protocol.

It is clear that modifying policy is useful and sometimes necessary. But how can policy changes be applied? As the above Ethereum example illustrates, they are easy to do with a hark fork. Indeed, until now, folklore wisdom suggested that any upgrades to the macroeconomics of a chain required a hard fork and were impossible to implement through a soft fork. After all, how can unupgraded miners accept blocks paying according to different rules? One instance illustrating the severity of the problem is a block paying out a higher reward in the new policy than what it used to pay in the old policy. It seems inherently difficult to achieve backwards compatibility when it comes to such drastic changes.

We put forth the first construction which allows policy changes through soft forks. In our path to doing so, we give a definition of what the macroeconomic policy is. Our construction is compatible with the current Ethereum blockchain and is implemented through a smart contract. The mechanism works for any changes in policy, as long as the new policy is *economically compatible* with the old one. It mandates that, in the long run, no more money can be generated by the new policy.

We illustrate the usefulness of our construction by presenting some applications. One notable application is a construction which patches the *bribery* attack on Non-Interactive Proofs of Proof-of-Work (NIPoPoWs). As the patch requires correct incentivization embeded in the consensus mechanism, it is impossible to apply without policy changes. Even though we do it through a soft fork, we are the first to propose a bribery-resilient variant of the NIPoPoWs protocol *in general*, as this attack remained an open problem before this work. The correction of this outstanding issue was an important motivation behind the present work.

Related work. Forking mechanisms in blockchains have been a topic of contention. A complete overview of hard forks and soft forks is given by Buterin [8] (who also presents some convincing arguments of why hard forks can sometimes be preferable). In addition to soft forks, velvet forks [9, 10] present an interesting, and softer, alternative, although they are not always possible and great care must be taken when adopting

them [11]. Even in the case of hard forks, core changes to consensus mechanisms must be deployed with prudence [12].

Smart contracts [13] were first used to distribute mining proceeds in the work of Luu et al. [14]. Our construction is inspired by their clever approach. Non-Interactive Proofs of Proof-of-Work (NIPoPoWs) were introduced by Kiayias et al. [9]. The bribing attack against them, which we patch in this work, was discovered by Bünz et al. [15]. Karantias et al. [16] perform convincing measurements that illustrate such attacks have not happened in the wild (yet).

Contributions. In this paper, we make the following key contributions:

- We formally define what a chain macroeconomic policy is. Our definition is generic and can be any algorithm that satisfies certain basic conditions.
- We define the notion of *economic compatibility* between policies, a necessary and sufficient condition for soft fork upgradability between policies.
- We put forth a generic mechanism for upgrading the macroeconomic policy of a chain through a *soft fork* and present it in the form of a *smart contract*.
- We present several applications of our scheme. Notably, we resolve the open problem of Non-Interactive Proof of Proof-of-Work (NIPoPoW) *bribability*.

2 Preliminaries

Blockchain systems maintain consensus through the dissemination of chains. A *chain* $\mathcal{C}$ is a finite sequence of *blocks*, and each block B is a triplet (x, s, ctr). A block id is the cryptographically secure hash of the triplet $H(x, s, \mathsf{ctr})$. Here, x denotes the set of confirmed messages (transactions), s denotes the block id of the previous block in the chain, and ctr denotes the *leader election information*, a nonce in the case of proof-of-work systems or a signature in the case of proof-of-stake systems. The first block in the chain is the genesis block $\mathcal{G}$ in which s is taken to be the empty string ϵ by convention. We write $\mathcal{C} \preccurlyeq \mathcal{C}'$ if $\mathcal{C}$ is a (not necessarily strict) prefix of $\mathcal{C}'$. We denote by $\mathcal{C}[i]$ the i^{th} block of the chain (zero-based). We use the Python range notation $\mathcal{C}[i{:}j]$ to denote the subsequence of blocks, or *subchain*, from i (inclusive) to j (exclusive). Omitting an index takes the subchain from the beginning or to the end respectively.

We denote $\mathcal{E}$ an *execution* of our blockchain protocol [17, 18]. The execution captures the messages exchanged by all parties throughout, as

well as the random coins produced during the mining process. We use κ to denote the security parameter.

The *block language* $\mathcal{L}_B$ is the set of all syntactically valid blocks and the *chain language* $\mathcal{L}_\mathcal{C} \subseteq \mathcal{L}_B^*$ is the set of all valid chains.

Recall that a blockchain can be upgraded with a *soft fork* or a *hard fork*. In both cases, the code of the node is modified and the new software is distributed to the users. Some of the users adopt the new code and those are known as *new* or *upgraded* miners. The ones that do not upgrade are the *old* or *unupgraded miners* who are running the old version of the node. Once downloaded, the new code is activated after a designated *activation block height*. The upgraded software contains new rules that govern the generation and validation of blocks. In both cases, the old rules are *not* forwards compatible with the new rules: If an old node generates an old-style block, it will not be validated by upgraded miners. In the case of a soft fork, the new rules are backwards compatible with the old rules: If a new node generates a new-style block, it will be validated by old miners. Provided the upgraded miners constitute a majority, unupgraded nodes will still follow the longest chain, which will contain only upgraded blocks. Their own blocks will be rejected, so they will be economically pushed to upgrade their software. However, in a hard fork, the new rules are *not* backwards compatible with the old rules: New nodes generate blocks that do not validate according to old rules. As such, the two populations create two distinct chains after the activation block height, which constitutes a chain fork. This is sometimes viewed as dangerous. Nevertheless, even in the case of hard forks, the old population typically eventually upgrades to the new rules and their temporary fork is abandoned.

3 Macroeconomic Policies

A chain policy defines how payouts are given to miners (or minters). While for simple policies it could be a constant, more complex policies may depend on the whole state $\mathcal{C}$ of the system.

Definition 1 (Macroeconomic Policy). *Let $\mathcal{L}_\mathcal{C}$ be the chain language. We call a function $\pounds : \mathcal{L}_\mathcal{C} \times \mathbb{N} \to \mathbb{R}^+$ of the system a* macroeconomic policy *if the function is efficiently computable and for every two chains $\mathcal{C}, \mathcal{C}'$ such that $\mathcal{C} \preccurlyeq \mathcal{C}'$, it holds that $\pounds(\mathcal{C}, i) \leq \pounds(\mathcal{C}', i)$. The system pays out an amount of $\pounds(\mathcal{C}, i)$ to the validator of the i^{th} block in $\mathcal{C}$.*

The above definition captures, quite generically, what the rewards of a miner can be. The requirement that the function is monotonic is

necessary, as it prescribes that money given out cannot be retroactively taken back. While the rewards can depend just on i, the ability of the function to inspect the whole chain $\mathcal{C}$ allows the system to employ complex rules. Additionally, note that it is possible that $|\mathcal{C}| > i$. In that case, the policy may decide to pay out rewards to a miner later during the system's execution. We allow the function to output *real* positive amounts payable, even though most systems employ integer outputs to avoid floating-point errors.

Let us look at a couple of typical policies for illustration purposes. Bitcoin's policy is a step function which began at 50 BTC per block and halves every 210,000 blocks. Additionally, rewards cannot be withdrawn for another $c = 100$ blocks, a constraint known as the *maturation time*:

$$\pounds_{\mathrm{BTC}}(\mathcal{C}, i) = \begin{cases} 0, \text{ if } |\mathcal{C}| + c < i \\ \frac{50}{2^j}, \text{ otherwise} \end{cases}, \text{ where } j = \lfloor \frac{i}{210000} \rfloor$$

Monero's policy emits money *smoothly*, which they argue [6] can help prevent infrastructural problems due to dramatic increases in hashrate when compared to Bitcoin's. Instead, they give out one 2^{18}th of their remaining money supply per block. Their maturation time is $c = 60$ blocks.

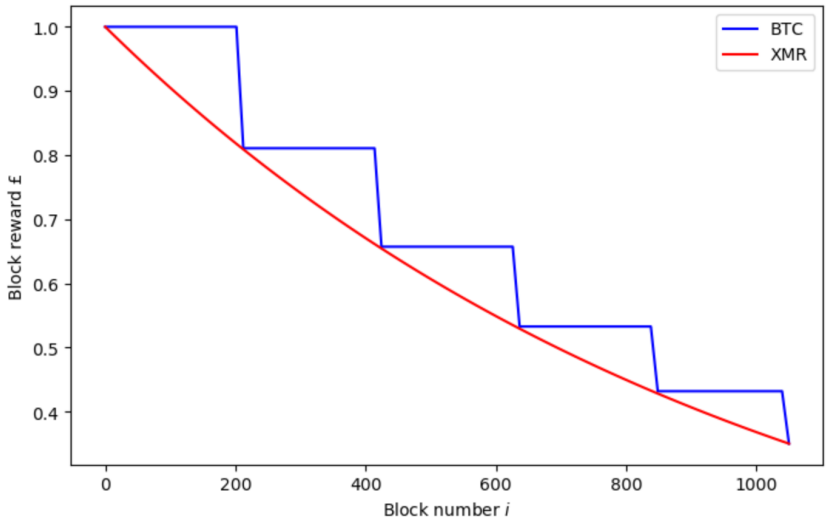
$$\pounds_{\mathrm{XMR}}(\mathcal{C}, i) = \begin{cases} 0, \text{ if } |\mathcal{C}| + c < i \\ \frac{2^{64} - 1 - \sum_{j=0}^{i-1} \pounds_{\mathrm{XMR}}(\mathcal{C}, j)}{2^{18}}, \text{ otherwise} \end{cases}$$

Note here that it so happens that an upgrade from Bitcoin's policy to Monero's policy could take place with a soft fork without any special mechanism, as long as Monero's supply is scaled appropriately to be upper bound by Bitcoin's at *every* block (see Figure 1).

In both systems, the payouts are deterministic (and in the *steady state* do not depend on $\mathcal{C}$), and the total supply at every point in time is $\sum_{i=0}^{|\mathcal{C}|} \pounds(\mathcal{C}, i)$. In fact, the total supply is bounded, and the bound is $\lim_{|\mathcal{C}| \to \infty} \sum_{i=0}^{|\mathcal{C}|} \pounds(\mathcal{C}, i)$. Some systems, such as DOGE do not have a maximum total supply and this limit diverges.

Ethereum's policy is a little trickier. A block can receive a reward even if it does not belong on the adopted chain. Instead, *uncle* blocks are rewarded, too [4,5]. In this system, the function $\pounds$ is defined on *blocktrees* instead of chains. The parameter i is generalized to denote any *path* in

Fig. 1. Bitcoin's staircase rewards compared to Monero's smooth emission with Bitcoin upper bounding Monero.



the blocktree, and the prefix notation $\preccurlyeq$ must, of course, be amended accordingly to mean *subgraph*. Any chain system whose consensus is based on a DAG [19,20] instead of a tree can be thus augmented. As long as the language $\mathbb{L}_{\mathcal{C}}$ is appropriately defined, our definition stands, albeit with a more complex interpretation. In this case, as the total supply depends on the execution (and in particular how many uncles it contains), it cannot be calculated exactly.

Let us now determine whether two policies are *backwards compatible*. We begin with a strict definition.

Definition 2 (Economic Compatibility). *A* new *policy* $\pounds'$ *is* backwards compatible *with an* old *policy* $\pounds$ with respect to chain $\mathcal{C}$ *if*

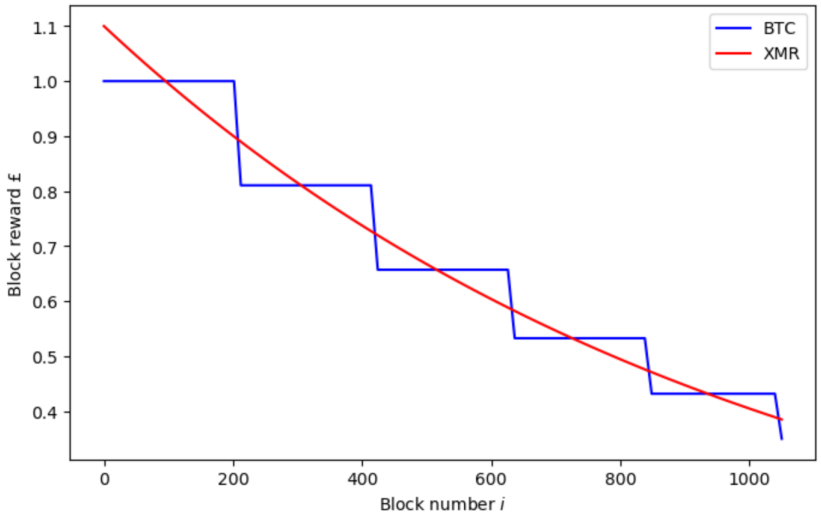
$$\sum_{i=0}^{|\mathcal{C}|-1} \pounds(\mathcal{C}, i) \geq \sum_{i=0}^{|\mathcal{C}|-1} \pounds'(\mathcal{C}, i)\,.$$

The two policies are backwards compatible *if they are compatible with respect to every chain.*

Compatibility mandates that the new policy does not pay out more money than the old policy for a particular chain. Note how this requirement is not made *for every block*, but instead *throughout history*. This leniency opens the door for upgrading to quite a wide range of policies.

For example, *for the same total supply*, Bitcoin's policy is backwards compatible with Monero's: Bitcoin begins by paying a smaller amount per block than Monero. This money is accumulated. After a while, Monero's supply has dropped and Bitcoin is paying more per block than Monero. However, the accumulated money can make up for the difference at every point in time (see Figure 2). Our construction in the next section will make it clear how this accumulation can take place in chains with smart contract support.

Fig. 2. Bitcoin's staircase rewards compared to Monero's smooth emission with the same total reward.



Generally, our policies will not be required to be *strictly* compatible. Instead, we will only mandate that they are compatible *eventually*.

Definition 3 (Eventual Compatibility). *A* new *policy* $\pounds'$ *is* eventually backwards compatible *with an* old *policy* $\pounds$ *with delay* d with respect to an execution $\mathcal{E}$ *if for every chain* $\mathcal{C}$ *adopted during the execution, for any* $i, j \in \mathbb{N}$ *such that* $0 \leq i + d < j < |\mathcal{C}|$*, there is a* $k \in \mathbb{N}$ *such that* $i \leq k < j$ *and* $\pounds'$ *is backwards compatible with* $\pounds$ *with respect to chain* $\mathcal{C}[:k]$.

Eventual backwards compatibility does not require that the new policy remains solvent *all of the time*. Instead, it promises that the miners will be paid out *eventually*, even though they may have to wait up to time d until the policy becomes solvent and has the money to pay them.

For example, *for the same given total supply*, Monero's policy is *eventually backwards compatible* with Bitcoin's: While initially Monero will require higher payments than Bitcoin, eventually Monero's smooth emission will reduce the supply sufficiently to drop below Bitcoin's reward (see Figure 2). After this, and when a certain number of blocks have been produced, a sufficient amount of money will have accumulated to be able to pay back the miners who mined more worthy Monero coins in the past. While the miners may have to wait for a delay d for the policy to achieve solvency, they will eventually be paid the correct amount.

To account for unlikely events, we relax this requirement further and require that eventual compatibility is only achieved *with high probability* in the security parameter (over the randomness of the execution). This relaxation enables us to build non-deterministic policies, as long as they are well-behaved with high probability.

It is useful to point out that backwards economic compatibility is a notion useful beyond soft and hard forks. It indicates that the new policy does not generate, in total, more money than the old policy. This implies that any previous assumption on upper bounds in supply is not violated (one such example is any total supply *firewall* limitation in the case of a sidechain [21]). Beyond a useful technical propertly, economic compatibility is primarily an *economic* assumption.

4 Construction

Our construction is based on a simple premise: Instead of paying out the miner directly, we can pay the proceeds of mining into a smart contract beneficiary. Before the soft fork begins, the smart contract is deployed on the old network. This deployment is verified by old and new miners alike. The soft fork then *mandates* that, after a particular activation block height, new blocks always pay into this smart contract's address. Blocks that do not pay into the designated address are rejected as invalid. Old miners accept new blocks because they just have a new valid beneficiary, and it so happens that it is the same for all the blocks they see, but without any notion of its semantics.

Once the beneficiary contract is deployed, it is responsible for managing the policy of the chain. It collects the proceeds of the old policy into its reserves and pays out the miners accordingly. If the new policy is backwards compatible with the old one with respect to every chain, the contract remains solvent. However, if the compatibility is only eventual, the contract will be insolvent at certain points in time. To address this,

it gives out *promises to pay* miners, which are assembled in a balance sheet, akin to an `ERC-20` [22] smart contract. This money can then be collected at a later time, when the contract becomes solvent again, which is guaranteed to occur after d blocks.

We illustrate our generic construction, parametrized by the new policy $\pounds$ in Algorithm 1. The contract is initially instantiated to an address as a regular contract. Anyone can do this, but it would typically be done by the cryptocurrency developers. The contract deployment address is noted and embeded into the code of the upgraded node software. The contract has three methods. The default method is paid out when the contract is designated as the beneficiary of any block. At this point, it is unclear what the identity of the miner is. The miner who actually produced the block places their identity in the form of their public key into a designated location within the block. In the case of Bitcoin soft forks, this is typically the *coinbase* transaction, but in a smart-contract-enabled blockchain, which our construction requires, this can be done more cleanly by the call to the identify method of our smart contract, in which the miner creates a transaction calling it and passing their public key pk as an argument to the call. The method call records the identity of the miner. Naturally, the miner must take care to drop from their block transactions of adversarial users calling the identify method, to avoid enterprising usurpers.

After the miner has identified themselves, they can claim the payout from the new policy by invoking the claim function. This function takes a block index i and evaluates the policy $\pounds$ on the current chain $\mathcal{C}$. Therefore it might need to be called at a later point by the miner to account for, say, maturation constraints. Note here that typically the policy will only depend on a small subset of the blocks in $\mathcal{C}$ and so not all of it needs to be evaluated. While some blockchains allow for access to past blocks liberally [23], the contract can replicate such behavior locally [24] if needed to recreate any portion of the chain required by the policy.

The function records the payout as delivered to the miner so that it cannot be doubly claimed. However, the payment is not actually delivered to the miner beyond a *promise to pay*, recorded in the balance sheet balances of the contract. If and when the contract becomes solvent, the miner can then call withdraw to get their money in the real native currency, such as Ether in our case. This behavior is similar to the balances maintained by an `ERC-20` contract. In fact, the beneficiary contract can be a fully fledged `ERC-20` contract, in which case the miners will be able to use their promise-to-pay tokens as if it were real ether.

Algorithm 1 The smart contract beneficiary which acts as a decentralized macroeconomic policy manager for policy $\pounds$.

```
1: contract Policy_£
2:     balances ← ∅
3:     claims ← ∅
4:     identities ← ∅
5:     payable function default()
6:         ▷ Collect proceeds from this block, but do not pay it out yet
7:     end function
8:     function identify(pk)
9:         identities[block.id] ← pk
10:    end function
11:    function claim(i)
12:        C ← get_chain()
13:        miner ← identities[i]
14:        v ← £(C, i)                    ▷ The particular policy is invoked at this point
15:        D ← v − claims[i]              ▷ D will be positive due to monotonicity of £
16:        claims[i] ← v
17:        balances[miner] ← balances[miner] + D        ▷ Create a promise to pay later
18:    end function
19:    function withdraw(v)                              ▷ An ERC-20-style withdrawal
20:        require(balances[msg.sender] ≥ v)
21:        require(address(this).balance ≥ v)           ▷ Ensure the contract is solvent
22:        balances[msg.sender] ← balances[msg.sender] − v
23:        msg.sender.send(v)
24:    end function
25: end contract
```

5 Blinded Mining

We have already seen that simple policies such as Bitcoin's and Monero's can be upgraded between one another. It should also be clear that increasing the reward maturation time is easily implementable.

One interesting and more complex policy involves requiring miners to generate and commit to a value χ during their block generation. The commitment $h = H(\chi)$ is placed in the block instead. The value χ is to be kept secret until after k blocks have passed, at which point the value should be revealed soon after, and certainly before $2k$ blocks have passed. This *blinded mining* process can be a useful tool for constructing consensus protocols that can withstand an adaptive adversary or suppression attacks [25].

A cryptoeconomic incentivization of the above protocol can be achieved with the following policy:

$$\mathcal{L}(\mathcal{C}, i) = \begin{cases} c & \text{if } \mathcal{C}[i{:}i+k] \text{ contain no } \chi \\ & \text{but } \mathcal{C}[i+k{:}i+2k] \text{ contains } \chi \\ (1+\epsilon)c & \text{if the above holds, and } \mathcal{C}[i] \text{ is the } \textit{first} \text{ block} \\ & \text{to reveal } \chi' \text{ for block } \mathcal{C}[i-j] \text{ with } j < k \\ 0 & \text{otherwise} \end{cases}$$

where $h = H(\chi), h' = H(\chi')$ are the commitments in $i, i-j$ respectively.

Here, the miner who mined the i^{th} block is rewarded with c only once they reveal the value χ. This revealing can be made in an appropriately structured transaction, even if they do not mine any future blocks. This requirements mandates a sort of *availability* by the miner: They are not paid until they reveal their committed value, and they must ensure they remain online to do so. Additionally, the miner must reveal it before $2k$ blocks, or else their rewards are gone. Lastly, if the value is leaked sooner, i.e., before k blocks have passed, to a different miner, that miner is rewarded with ϵc extra rewards, in addition to their c that they receive for playing fairly. This $0 < \epsilon < 1$ *slashes* the miner who revealed the value too soon. The value ϵ must be large enough ($0 < \epsilon$) to incentivize competing miners to find the value and reveal it sooner, but small enough to incentivize the miner of i to keep the value secret ($\epsilon \ll 1$ to account for the *time value of money*).

6 Unbribability

A notable achievement possible with a policy upgrade is making NIPoPoWs unbribable. While the precise details of the NIPoPoW protocol are beyond the scope of this work, let us review the essential parts here to motivate the discussion.

In a proof-of-work chain, a block B satisfies the proof-of-work equation $H(B) \leq T$, where T denotes the *mining target* (this can be a constant or a variable). Some blocks satisfy this equation better than others, and specifically achieve $H(B) \leq \frac{T}{2^\mu}$ for some $\mu \in \mathbb{N}$. Such blocks are called μ-superblocks (or superblocks of level μ).

The NIPoPoW protocol posits that a *superlight client*, which functions as an SPV node to the blockchain, can synchronize from a full node by receiving only a small *sample* of superblocks. More concretely, if the full node presents a subsequence of $m \in \mathbb{N}$ superblocks of level μ, then the superlight client is convinced that approximately $m2^\mu$ regular blocks

exist in the underlying chain, but these do not need to be sent over the network. Leveraging this basic clever idea, the protocol achieves an exponential improvement in communication complexity compared to legacy SPV clients [9]. The parameter m has a minimum value, but can be increased as needed to ensure security (with a corresponding performance penalty).

A block is a μ-superblock with probability $2^{-\mu}$, so they are exceedingly rare as μ increases. Unfortunately, they are rewarded only as much as regular blocks. As such, an adversary can cheaply *bribe* miners to keep such blocks secret [15]. In this attack, the adversary requests that the miners never broadcast these blocks into the network, and pays the miners behind-the-scenes in exchange for this commitment. In fact, such bribes can even be written in the form of a smart contract, completely removing the need for the adversary and the miners to maintain rogue offchain communication channels. While honest miners will not succumb to such behavior, rational miners might. A rational adversary is also incentivized to give out such bribes if they wish to convince a superlight client that a large amount of money has been transferred to them (the exact amount can be calculated using the methods of Bonneau et al. [26]).

We now put forth a method for defending against this attack. The attack becomes uneconomical if the reward schedule of the chain is modified so that a μ-superblock's worth is proportional to the amount of underlying blocks it captures. More precisely, each μ-superblock must be worth 2^{μ} more than a regular block, provided at least m superblocks of level μ have appeared on the network. In this case, bribing to suppress superblocks capturing a certain amount of proof-of-work requires the same bribe as suppressing the whole underlying chain. As long as such bribes are not economical (an assumption required for the blockchain to function), superblock bribes are not economical either.

To make this new policy backwards compatible with the old policy, the value of regular blocks must be reduced. But how should we ascribe value to these blocks? Suppose the old policy pays out 1 unit of currency per block. If a regular block (which is not a superblock) pays out a value of c and a μ-superblock (which is not a $\mu+1$ level superblock) pays out a value of $2^{\mu}c$, then we are led to the following paradox: The expected value of the reward diverges:

$$\mathbb{E}[\mathcal{L}(\mathcal{C}, i)] = \sum_{\mu=0}^{\infty} c2^{\mu+1} Pr[H(B) \leq \frac{T}{2^{\mu}}] = \sum_{\mu=0}^{\infty} c2^{\mu+1}2^{-\mu} = \infty$$

However, the probability of such divergence is negligible. We will make use of this fact to construct a policy that is eventually backwards compatible with the constant policy with overwhelming probability in the security parameter κ. The policy progresses in epochs. In each epoch j, a constant c_j is adopted as the reward of a 0-level block. Within each epoch, the invariant that each block of level μ is worth 2^μ more than each regular block is maintained, i.e., $\pounds(\mathcal{C}, i_1) = 2^\mu \pounds(\mathcal{C}, i_2)$ where $\mathcal{C}[i_1]$ is a regular block and $\mathcal{C}[i_2]$ is a μ-superblock (however, this invariant is not maintained *across* epochs). We will now define the lengths of these epochs and the value c_j.

Consider a chain $\mathcal{C}$ with length $|\mathcal{C}|$, a superblock level μ, and a constant m. Observe that the number X of μ-superblocks appearing in $\mathcal{C}$ follows a binomial distribution with a Bernoulli probability of success $p = 2^{-\mu}$ and $|\mathcal{C}|$ trials. As such, $\mathbb{E}[X] = 2^{-\mu}|\mathcal{C}|$. We can now examine whether at least m superblocks of some level μ have appeared in this chain. Call this event DEFAULT. We want our system to avoid this event, as it will imply that our policy will become insolvent. Let us consider the case when $\mathbb{E}[X] < m$, and so we do not expect the bad event to occur. Still, we wish for the probability of the event occurring to be negligible. Let δ be the value such that $m = (1+\delta)2^{-\mu}|\mathcal{C}|$, i.e., $\delta = \frac{m}{2^{-\mu}|\mathcal{C}|} - 1 > 0$.

Since the trials are mutually independent Bernoulli trials, we can apply a Chernoff bound to obtain:

$$Pr[\text{DEFAULT}] = Pr[X \geq m] = Pr[X \geq (1+\delta)\,\mathbb{E}[X]] < e^{-\delta^2 \frac{\mathbb{E}[X]}{3}}$$

When does this probability attain a value negligible in the security parameter κ? We have:

$$e^{-\delta^2 \frac{\mathbb{E}[X]}{3}} \leq 2^\kappa \Leftrightarrow \frac{\delta^2 2^{-\mu}|\mathcal{C}|}{3} \lg e \geq \kappa$$

Replacing δ with its value, we obtain the following sufficient condition for solvency:

$$\begin{aligned}
&(\frac{m}{2^{-\mu}|\mathcal{C}|} - 1)^2 \frac{2^{-\mu}|\mathcal{C}|}{3} \lg e \geq \kappa \\
\Leftrightarrow &\frac{m^2}{2^{-\mu}} + 2^{-\mu}|\mathcal{C}| \geq \frac{3\kappa}{\lg e} + 2m \\
\Leftarrow &\frac{m^2}{2^{-\mu}|C|} \geq \frac{3\kappa}{\lg e} + 2m
\end{aligned}$$

$$\Leftrightarrow 2^{-\mu}|C| \leq \frac{m^2}{\frac{3\kappa}{\lg e} + 2m}\,.$$

Observe that the right-hand side is a constant, call it ζ. We can therefore be certain with overwhelming probability in κ that superblocks of levels μ or higher will not appear in chains of length $|\mathcal{C}|$ or less. This immediately leads to an algorithm for epoch evolution: Begin at epoch $j = 1$ in which the reward is $\frac{c_1}{2}$. As long as our chain $|\mathcal{C}|$ has size below $2^\mu\zeta$, we treat our system as if superblocks of level μ and above will never appear. Blocks of levels $0, 1, \cdots, \mu - 1$ receive pay out rewards of $c_j, 2c_j, \cdots, 2^{\mu-1}c_j$. The expected reward per block in this epoch is $\mathbb{E}[\mathcal{L}_j] = \sum_{i=0}^{\mu-1} c_j 2^i / 2^{i+1} = \frac{jc_j}{2}$. Whenever a chain size of $2^\mu\zeta$ is reached, the epoch advances to $j + 1$ and the reward is updated so that $\mathbb{E}[\mathcal{L}_{j+1}] = \mathbb{E}[\mathcal{L}_j]$. Solving for c_{j+1}, the new reward at level 0 then becomes $c_{j+1} = \frac{j}{j+1}c_j$. As you can see, these do not change the reward by much, and the update happens exponentially more rarely as time goes by.

The above construction lets us state the following lemma:

Lemma 1 (Compatibility of NIPoPoW rewards). *The policy $\mathcal{L}$ described above is eventually backwards compatible with a policy of constant rewards of amount $(1 + \epsilon)\frac{c_1}{2}$ with overwhelming probability in κ.*

Proof (Sketch). The proof is immediate from the above construction. Each epoch j with maximum chain length $|\mathcal{C}|$ maintains an expected payout per block which is $\mathbb{E}[\mathcal{L}_j] = \frac{c_1}{2}$. This is ensured with overwhelming probability in κ, as it was argued through the above Chernoff bound that superblocks of level $\mu \geq \frac{\zeta}{|C|}$ appear with only negligible probability. Applying a union bound over all epochs ensures a negligible probability of failure overall in the parameter $\kappa - \log(L)$ where L denotes the total execution time (as the number of epochs grows logarithmically in L). Each block reward is independent from the rest. The deposits available to the policy are the sum of these rewards and are bounded by a Chernoff bound. Thus, this sum will converge with high probability to its expectation after a sufficient number of blocks d. As $(1 + \epsilon)\frac{c_1}{2} > \frac{c_1}{2}$ this ensures eventual compatibility. The delay d depends on the choice of the parameter ϵ. A tradeoff exists between lowering the reward slightly to ensure eventual compatibility more quickly. □

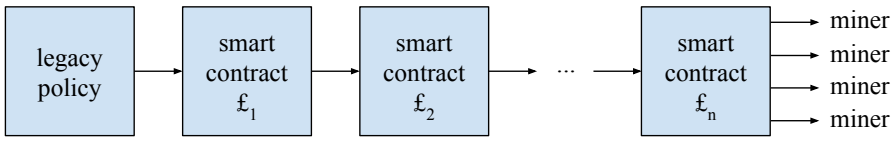
7 More Complex Upgrades

Multiple Upgrades. One outstanding question is how to apply a policy upgrade $\mathcal{L}''$ on top of a chain in which a policy upgrade $\mathcal{L}'$ has already

been applied; that is, how to apply multiple policy upgrades in series. The solution is to employ *yet another* smart contract as a second intermediate step between the block reward and the miner.

The approach works by having $\pounds'$ manage the first policy by receiving the money of the legacy policy $\pounds$ as before. When the time comes to call the identify function, the second soft fork requires the pk provided to be the second smart contract to which $\pounds''$ is deployed. As such, the first smart contract always pays out into the second. The miner variable in any valid execution of the first smart contract always takes on the address of the second smart contract after the second soft fork. The second smart contract can then have its own identify function, which uses a different portion of the block to identify the final receipient. The technique can be used repeatedly in series, as illustrated in Figure 3.

Fig. 3. A series of upgrades $\pounds_1, \cdots, \pounds_n$. Each of the contracts maintains its own balance and pays into the next. Only the final contract distributes proceeds to the true miners.



More Complex Policies. For simplicity, our definition of the policy $\pounds(\mathcal{C}, i)$ returns the amount payable to the miner who mined the block $\mathcal{C}[i]$. It is possible to devise of more complex policies in which the policy pays out multiple people per block or in general does not have just one receipient per block. One such example is a policy that distributes payments to miners who are mining blocks often. It is easy to generalize the definition of policies to allow for such a scenario. The function is defined to be $\pounds(\mathcal{C})$ and returns a dictionary mapping from address to amount payable. The monotonocity condition is then the obvious generalization of our previous condition: Given two chains $\mathcal{C} \preccurlyeq \mathcal{C}'$, the keys in the dictionary $\pounds(\mathcal{C})$ must be a subset of the keys in the dictionary $\pounds(\mathcal{C}')$. Additionally, for every key in both dictionaries, the value in $\pounds(\mathcal{C})$ must be smaller than or equal to the value in $\pounds(\mathcal{C}')$.

References

1. Nakamoto S.: Bitcoin: A peer-to-peer electronic cash system. Available at: https://bitcoin.org/bitcoin.pdf: URL https://bitcoin.org/bitcoin.pdf (2008)
2. Dwork C., Naor M.: Pricing via processing or combatting junk mail. In Annual International Cryptology Conference: pp. 139–147: Springer (1992)
3. Kiayias A., Russell A., David B., Oliynykov R.: Ouroboros: A Provably Secure Proof-of-Stake Blockchain Protocol. In J. Katz, H. Shacham (editors), Annual International Cryptology Conference: vol. 10401 of *LNCS*: pp. 357–388: Springer: Springer (2017)
4. Kiayias A., Panagiotakos G.: Speed-Security Tradeoffs in Blockchain Protocols. IACR Cryptol. ePrint Arch.: vol. 2015, p. 1019 (2015)
5. Kiayias A., Panagiotakos G.: On trees, chains and fast transactions in the blockchain. In International Conference on Cryptology and Information Security in Latin America: pp. 327–351: Springer (2017)
6. Van Saberhagen N.: CryptoNote v2.0. Available at: https://cryptonote.org/whitepaper.pdf: URL https://cryptonote.org/whitepaper.pdf (2013)
7. Schoedon A.: EIP-1234: Constantinople Difficulty Bomb Delay and Block Reward Adjustment. URL: https://eips.ethereum.org/EIPS/eip-1234 (2018)
8. Buterin V.: Hard Forks, Soft Forks, Defaults and Coercion. URL http://web.archive.org/web/20080207010024/http://www.808multimedia.com/winnt/kernel.htm (2017)
9. Kiayias A., Miller A., Zindros D.: Non-Interactive Proofs of Proof-of-Work. In International Conference on Financial Cryptography and Data Security: Springer (2020)
10. Zamyatin A., Stifter N., Judmayer A., Schindler P., Weippl E., Knottenbelt W., Zamyatin A.: A Wild Velvet Fork Appears! Inclusive Blockchain Protocol Changes in Practice. In International Conference on Financial Cryptography and Data Security: Springer (2018)
11. Kiayias A., Polydouri A., Zindros D.: The Velvet Path to Superlight Blockchain Clients. IACR Cryptology ePrint Archive: URL http://eprint.iacr.org/2020/1122 (2020)
12. Ciampi M., Karayannidis N., Kiayias A., Zindros D.: Updatable Blockchains. In European Symposium on Research in Computer Security: pp. 590–609: Springer (2020)
13. Buterin V., et al.: A next-generation smart contract and decentralized application platform. white paper (2014)
14. Luu L., Velner Y., Teutsch J., Saxena P.: Smartpool: Practical decentralized pooled mining. In 26th USENIX Security Symposium (USENIX Security 17): pp. 1409–1426 (2017)
15. Bünz B., Kiffer L., Luu L., Zamani M.: Flyclient: Super-Light Clients for Cryptocurrencies. (2020)
16. Karantias K., Kiayias A., Zindros D.: Compact Storage of Superblocks for NIPoPoW Applications. In The 1st International Conference on Mathematical Research for Blockchain Economy: Springer Nature (2019)
17. Garay J. A., Kiayias A., Leonardos N.: The Bitcoin Backbone Protocol: Analysis and Applications. In E. Oswald, M. Fischlin (editors), EUROCRYPT 2015, Part II: vol. 9057 of *LNCS*: pp. 281–310: Springer, Heidelberg: doi:10.1007/978-3-662-46803-6_10: updated version at http://eprint.iacr.org/2014/765. (2015)

18. Garay J. A., Kiayias A., Leonardos N.: The Bitcoin Backbone Protocol with Chains of Variable Difficulty. In J. Katz, H. Shacham (editors), CRYPTO 2017, Part I: vol. 10401 of *LNCS*: pp. 291–323: Springer, Heidelberg (2017)
19. Sompolinsky Y., Lewenberg Y., Zohar A.: SPECTRE: A Fast and Scalable Cryptocurrency Protocol. IACR Cryptology ePrint Archive: vol. 2016: URL `http://eprint.iacr.org/2016/1159`
20. Sompolinsky Y., Wyborski S., Zohar A.: PHANTOM and GHOSTDAG: A Scalable Generalization of Nakamoto Consensus. IACR Cryptology ePrint Archive: URL `http://eprint.iacr.org/2018/104` (2018)
21. Kiayias A., Gaži P., Zindros D.: Proof-of-Stake Sidechains. In IEEE Symposium on Security and Privacy: IEEE: IEEE (2019)
22. Vogelsteller F., Buterin V.: ERC-20 Token Standard. Sept. 2017. URL: https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20-tokenstandard.md (2015)
23. Buterin V.: Blockhash refactoring. URL: `https://github.com/ethereum/EIPs/blob/master/EIPS/eip-210.md` (2017)
24. Karantias K., Kiayias A., Zindros D.: Smart Contract Derivatives. In The 2nd International Conference on Mathematical Research for Blockchain Economy: Springer Nature (2020)
25. Anonymized: Decentralized Blockchain Interoperability. Ph.D. thesis (2020)
26. Bonneau J., Clark J., Goldfeder S.: On Bitcoin as a public randomness source. IACR Cryptology ePrint Archive: vol. 2015: URL `https://eprint.iacr.org/2015/1015.pdf` (2015)