SmartPool: Practical Decentralized Pooled Mining

Loi Luu
National University of Singapore
loiluu@comp.nus.edu.sg

Yaron Velner
The Hebrew University of Jerusalem
yaron.welner@mail.huji.ac.il

Jason Teutsch
TrueBit Foundation
jt@truebit.io

Prateek Saxena
National University of Singapore
prateeks@comp.nus.edu.sg

Abstract

Cryptocurrencies such as Bitcoin and Ethereum are operated by a handful of mining pools. Nearly 95% of Bitcoin's and 80% of Ethereum's mining power resides with less than ten and six mining pools respectively. Although miners benefit from low payout variance in pooled mining, centralized mining pools require members to trust that pool operators will remunerate them fairly. Furthermore, centralized pools pose the risk of transaction censorship from pool operators, and open up possibilities for collusion between pools for perpetrating severe attacks.

In this work, we propose SmartPool, a novel protocol design for a decentralized mining pool. Our protocol shows how one can leverage *smart contracts*, autonomous blockchain programs, to decentralize cryptocurrency mining. SmartPool gives transaction selection control back to miners while yielding low-variance payouts. SmartPool incurs mining fees lower than centralized mining pools and is designed to scale to a large number of miners. We implemented and deployed a robust SmartPool implementation on the Ethereum and Ethereum Classic networks. To date, our deployed pools have handled a peak hashrate of 30 GHs from Ethereum miners, resulting in 105 blocks, costing miners a mere 0.6% of block rewards in transaction fees.

1 Introduction

Cryptocurrencies such as Bitcoin and Ethereum offer the promise of a digital currency that lacks a centralized issuer or a trusted operator. These cryptocurrency networks maintain a distributed ledger of all transactions, agreed upon by a large number of computation nodes (or *miners*). The most widely used protocol for agreement is Nakamoto consensus, which rewards one miner every epoch (lasting, say, 10 minutes as in Bitcoin) who exhibits a solution to a probabilistic computation puzzle called a "proof-of-work" (or PoW) puzzle [1]. The winning miner's solution includes a transaction *block*, which is appended to the distributed ledger that all miners maintain. The reward is substantial (e.g. 12.5 BTC in Bitcoin, or $30,000$ USD at present), incentivizing participation.

Nakamoto-based cryptocurrencies, such as Bitcoin and Ethereum, utilize massive computational resources for their mining. Finding a valid solution to a PoW puzzle is a probabilistic process, which follows a Poisson distribution, with a miner's probability of finding a solution within an epoch determined by the fraction of computation power it possesses in the network. Miners with modest computational power can have extremely high variance. A desktop CPU would mine 1 Bitcoin block in over a thousand years, for instance [2]. To reduce variance, miners join *mining pools* to mine blocks and share rewards together. In a mining pool, a designated pool *operator* is responsible for distributing computation sub-puzzles of lower difficulty than the full PoW block puzzle to its members. Each solution to a sub-puzzle has a probability of yielding a solution to the full PoW block puzzle—so if enough miners solve them, some of these solutions are likely to yield blocks. When a miner's submitted solution yields a valid block, the pool operator submits it to the network and obtains the block reward. The reward is expected to be fairly divided among all pool members proportional to their contributed solutions.

Problem. Centralized pool operators direct the massive computational power of their pools' participants. At the time of this writing, Bitcoin derives at least 95% of its mining power from only 10 mining pools; the Ethereum network similarly has 80% of its mining power emanating from 6 pools. Previous works have raised concerns about consolidation of power on Bitcoin [3,4]. Recent work by Apostolaki *et al.* has demonstrated large-scale network attacks on cryptocurrencies, such as double spending and network partitioning, which exploit centralized mining status quo [5]. By design, if a single pool operator controls more than half of the network's total mining power, then a classical 51% attack threatens the core security of the Nakamoto consensus protocol [1]. Cryptocurrencies have witnessed that a single pool has commandeered more than half of a cryptocurrency's hash rate (*e.g.* `DwarfPool`[1] in Ethereum and `GHash.io`[2] in Bitcoin) on several occasions. In such cases, the pool operator's goodwill has been the only barrier to an attack.

Furthermore, pools currently dictate which transactions get included in the blockchain, thus increasing the threat of transaction censorship significantly [6]. While some Bitcoin pools currently offer limited control to miners of transaction selection via the getblocktemplate protocol [7], this protocol only permits a choice between mining with a transaction set chosen by the pool or mining an empty block. The situation is worse in Ethereum where it is not yet technically possible for miners in centralized pools to reject the transaction set selected by the operator. For example, users recently publicly speculated that a large Ethereum pool favored its own transactions in its blocks to gain an advantage in a public crowdsale [3].

One can combat these security issues by running a pool protocol with a decentralized network of miners in place of a centralized operator. In fact, one such solution for Bitcoin, called P2POOL [8], already exists. However, P2POOL has not attracted significant participation from miners, and consequently its internal operational network remains open to infiltration by attackers. Secondly, technical challenges have hindered widespread adoption. Scalable participation under P2POOL's current design would require the system to check a massive number of sub-puzzles. Furthermore, P2POOL only works for Bitcoin; we are not aware of any decentralized mining approach for Ethereum.

Solution. This work introduces a new and practical solution for decentralized pooled mining called SMARTPOOL. We claim two key contributions. First, we observe that it is possible to run a decentralized pool mining protocol as a *smart contract* on the Ethereum cryptocurrency. Our solution layers its security on the existing mining network of a large and widely deployed cryptocurrency network, thereby mitigating the difficulty of bootstrapping a new mining network from scratch. Secondly, we propose a design that is efficient and scales to a large number of participants. Our design uses a simple yet powerful probabilistic verification technique which guarantees the fairness of the payoff. We also introduce a new data structure, the *augmented Merkle tree*, for secure and efficient verification. Most importantly, SMARTPOOL allows miners to freely select which transaction set they want to include in a block. If widely adopted, SMARTPOOL makes the underlying cryptocurrency network much more censorship-resistant. Finally, SMARTPOOL does not charge any fees [4], unlike centralized pools, and disburses all block rewards to pool participants entirely.

SMARTPOOL can be used to run mining pools for several different cryptocurrencies. In this work, we demonstrate concrete instantiations for Bitcoin and Ethereum. SMARTPOOL can be run natively within the protocol of a cryptocurrency — for instance, it can be implemented in Ethereum itself. We believe SMARTPOOL can support a variety of standard payoff schemes, as in present mining pools. In this work, we demonstrate the standard pay-per-share (or PPS) scheme in our implementation. Supporting other standard schemes like pay-per-last-n-shares (PPLNS) and schemes that disincentivize against block withholding attacks [9–11] is left for future work.

Results. We have implemented SMARTPOOL prototypes for Bitcoin and Ethereum mining and deployed them on Ethereum's testnet. We measure the costs for miners when submitting their contributions to the pool. Our experiments show that these operating costs are negligible compared to the projected income from block rewards (*e.g.* less than 1% of gross income) for both Bitcoin and Ethereum. This cost is lower than fees offered by centralized pools. Furthermore, each miner has to send only a few messages per day to SMARTPOOL. Finally, although being decentralized, SMARTPOOL still offers the advantage of low variance payouts like centralized pools.

A SMARTPOOL implementation has been released and deployed on the main network via a crowd-funded community project [12]. As of 18 June 2017, SMARTPOOL-based pools have mined in total 105 blocks on both Ethereum and Ethereum Classic networks and have successfully handled a peak hashrate of 30 GHs from 2 substantial miners. SMARTPOOL costs miners as little as 0.6% for operational transaction fees, which is much less than 3% fees taken in centralized pools like F2Pool [5].

As a final remark, SMARTPOOL does not make centralized pooled mining in cryptocurrencies impossible, nor does it incentivize against centralized mining or alter the underlying proof-of-work protocol (as done in work by Miller *et al.* [13]). SMARTPOOL simply offers a practical alternative for miners to move away from centralized pools without degrading functionality or rewards.

Contributions. We claim the following contributions:

- We introduce a new and efficient decentralized pooled mining protocol for cryptocurrencies. By

[1] https://forum.ethereum.org/discussion/5244/dwarfpool-is-now-50-5

[2] https://www.cryptocoinsnews.com/warning-ghash-io-nearing-51-leave-pool/

[3] https://www.reddit.com/r/ethereum/comments/6itye9/collecting_information_about_f2pool/

[4] The caveat here is that cryptocurrency miners will pay Ethereum transaction fees to execute SMARTPOOL distributively.

[5] https://www.f2pool.com/ethereum-blocks

leveraging smart contracts in existing cryptocurrencies, a novel data structure, and an efficient verification mechanism, SMARTPOOL provides security and efficiency to miners.

- We implemented SMARTPOOL and deployed real mining pools on Ethereum and Ethereum Classic. The pools have so far mined 105 real blocks and have handled significant hashrates while deferring only 0.6% of block rewards to transaction fee costs.

2 Problem and Challenges

We consider the problem of building a decentralized protocol which allows a large open network to collectively solve a computational PoW puzzle, and distribute the earned reward between the participants proportional to their computational contributions. We expect such a protocol to satisfy the following properties:

- *Decentralization.* There is no centralized operator who operates the protocol and manage other participants. The protocol is collectively run by all participants in the network. There is also no requirement for joining, *i.e.* anyone with sufficient computation power can freely participate in and contribute to solving the PoW puzzle.
- *Efficiency.* The protocol running costs should be low and offer participants comparable reward and low variance guarantees as centralized operations. Furthermore, communication expenses, communication bandwidth, local computation and other costs incurred by participants must be reasonably small.
- *Security.* The protocol protects participants from attackers who might steal rewards or prevent others from joining the protocol.
- *Fairness.* Participants receive rewards in proportion to their share of contributions.

In this paper we focus on this list of properties with respect to mining pools. Cryptocurrencies like Bitcoin and Ethereum reward network participants (or miners) new crypto-coins for solving computationally hard puzzles (or proof-of-work puzzles) [1,14,15]. Typically, Bitcoin miners competitively search for a `nonce` value satisfying

$$\mathsf{H}(\texttt{BlockHeader} \,||\, \texttt{nonce}) \leq D \qquad (1)$$

where H is some preimage-resistant cryptographic hash function (*e.g.* SHA-256), `BlockHeader` includes new set of transactions that the miner wants to append to the ledger and D is a global parameter which determines the puzzle hardness. Ethereum uses a different, ASIC-resistant PoW function [16]. which requires miners to have a (predetermined) big dataset of 1 GB (increasing over time). Thus, in Ethereum, the condition (1) becomes

$$\mathsf{H}(\texttt{BlockHeader} \,||\, \texttt{nonce} \,||\, \texttt{dataset}) \leq D$$

in which the `dataset` includes 64 elements of the 1GB dataset that are randomly sampled with the `nonce` and the `BlockHeader` as the random seed.

Finding a solution for a PoW puzzle in cryptocurrencies requires enormous amount of computation power. Thus miners often join resources and solve the puzzle together via pooled mining. Currently, most mining pools follow a centralized approach in which an operator manages the pool and distributes work to pool miners. Here we are interested in a decentralized approach that allows miners to collectively run and manage the pool without inherent trust in any centralized operator.

Threat model and security assumptions. Cryptocurrencies like Bitcoin and Ethereum allow users to use pseudonymous identities in the network. Users do not have any inherent identities and there is no PKI in the network. Our solution adheres to this setting.

We consider a threat model where miners are *rational*, which means they can deviate arbitrarily from the honest protocol to gain more reward. An alternative is a *malicious* model where the attacker does anything just to harm other miners. In this work, we are not interested in the malicious model since i) such sustained attacks in cryptocurrencies often require huge capital, and ii) existing centralized pools are not secure in such a model either [9–11]. We also assume that the adversary controls less than 50% of the computation power in the network on which SMARTPOOL runs. This assumption rules out double-spending via 51% attacks [1].

On the other hand, we do not make any assumption on the centralization or trusted setup in our solution apart from what have been made in existing cryptocurrencies [6].

2.1 Existing Solutions

In the widely adopted centralized pooled mining protocol, there is a pool operator who asks pool miners to solve pool sub-puzzles by finding `nonce` so that the hash satisfies some smaller difficulty d ($d \ll D$). A solution for a pool-puzzle is called a *share* and has some probability of being a valid solution for the main PoW puzzle. Once a miner in the pool finds a valid block, the reward, i.e., new crypto-coins, is split among all pool miners in proportion to the number of their valid submitted shares [2].

Despite being widely used in practice, centralized mining pools have several problems including network

[6] Bitcoin and Ethereum have trusted setups where the first blocks are constructed and provided by Satoshi Nakamoto (for Bitcoin) and Ethereum Foundation (for Ethereum).

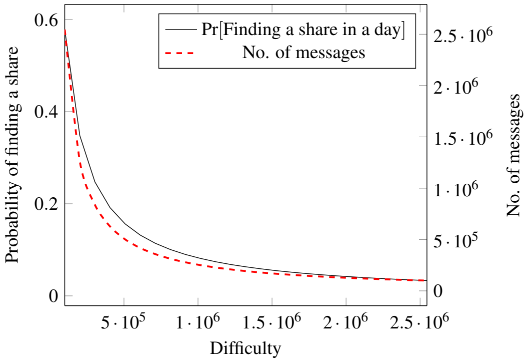

Figure 1: The effect of share's difficulty on i) the probability of a miner with 1 GHs finding a share within a day as per [2]; ii) resource (*i.e.* number of messages) consumed by a miner; in a decentralized mining pool for Bitcoin (*e.g.* P2Pool).

centralization and transaction censorship as discussed in Section 1. P2Pool for Bitcoin is the first and only deployed solution we are aware of which decentralizes pooled mining for cryptocurrencies [8]. At a high level, P2Pool decides on the contribution of each miner by running an additional Nakamoto Consensus protocol to build a *share-chain* between all miners in the pool. The share-chain includes all shares submitted to the pool, one after another (akin to the normal Bitcoin blockchain, but each block is a share). To guarantee that each share is submitted and credited exactly once, P2Pool leverages coinbase transactions, which are special transactions that pay block reward to miners (see details in Section 3.3).

P2Pool satisfies almost all ideal properties of a decentralized pool (defined in Section 2) except the *efficiency* and *security* properties. Specifically, P2Pool entails a high performance overhead since the number of messages exchanged between miners is a scalar multiple of the number of shares in the pool. When the share difficulty is low, miners have to spend a lot of resources (*e.g.* bandwidth, local computation) to download, and verify each other's shares. Figure 1 demonstrates how adjusting the difficulty of shares affects the variance of miners' reward and the amount of resource (both bandwidth and computation) consumed per miner (with $1GHs$ capacity) in a decentralized pool like P2Pool. As a result, P2Pool requires high share difficulty in order to reduce the number of transmitted messages. Therefore P2Pool miners experience higher reward variance than they would when mining with centralized pools. As discussed in [2], high variance in the reward (*i.e.* the supply of money) decreases miners' utility by making it harder for them to predict their income and verify that their systems are working correctly. Perhaps as a result, P2Pool has to date attracted only a few miners who comprise a negligible fraction of Bitcoin mining power (as of June 23, 2017, the last block mined by P2Pool was 22 days ago [8]).

The security of P2Pool's share-chain depends on the amount of computation power in its pool. As of this writing, P2Pool accounts for less than 0.1% of Bitcoin mining power, thus P2Pool's share chain is vulnerable to 51% attacks from adversaries who control only 0.1% of Bitcoin mining power. Hence P2Pool may not offer better security guarantees than centralized pools.

2.2 Our Solution and Challenges

Our solution for a decentralized pooled mining leverages Ethereum smart contracts which are decentralized autonomous agents running on the blockchain itself [17, 18]. A non-contract account has an address and balance in Ether, the native currency for Ethereum. A smart contract has, in addition, code and private persistent storage (*i.e.* a mapping between variables and values). Smart contract code is akin to a normal program which can manipulates stored variables. To invoke a contract (*i.e.* execute its code) at address *addr*, users send a transaction to *addr* with an appropriate payload, *i.e.* payment for the execution (in Ether) and/or input data for the invocation. The contract code executes correctly on the blockchain as long as a majority of Ethereum miners faithfully follow the Ethereum protocol.

At a high level, SmartPool replaces the mining pool operator with a smart contract. The smart contract acts as a trustless bookkeeper for the pool by storing all shares submitted by miners. When a new share is submitted, the contract verifies the validity of the share, checks that no previous record of the share exists, and then updates the corresponding miner's record. We allow miners to locally generate the block template of the pool (discussed more in Section 3.3). If a miner finds a share which is a valid block, it will broadcast the block to the cryptocurrency network, the reward will be instantly credited to SmartPool. SmartPool then disburses the block reward fairly to all miners in the pool.

Challenges. There are several challenges in building such a smart contract for a mining pool. We illustrate them by considering a straw-man solution (called `StrawmanPool`) in Figure 2 which implements a decentralized pool as a Ethereum smart contract. The solution works by having a smart contract which receives all the shares submitted by miners, verifies each of them and records number of shares one has submitted. The contract has a designated address for receiving block reward. A share is valid if it uses the contract address as the coinbase address (*i.e.*, the address that the block reward is sent to) and satisfies the predefined difficulty (*e.g.* Line 6). On each share submission, the pool verifies the share and updates the contribution statistics of the pool members (Line 13). If a miner finds a valid block, the smart

```
 1 contract StrawmanPool{
 2   mapping (uint256 => boolean) mSubmittedShares;
 3   mapping (uint256 => int) mContribution;
 4   function submitShare(someShare) returns (boolean){
 5     // check validity
 6     if !isValid(someShare)
 7       return false;
 8     // check if the share has been submitted
 9     if mSubmittedShares[someShare.hash]
10       return false;
11     mSubmittedShares[someShare.hash] = true;
12     // update miner's contribution
13     mContribution[msg.owner] += 1;
14     // distribute reward if is a valid block
15     if isValidBlock(someShare)
16       distributeReward(mContribution);
17     return true;
18   }}
```

Figure 2: Pseudo-code of a straw-man solution which implements a mining pool in a smart contract.

contract distributes the reward to miners in the pool proportional to their contribution by using any of the standard payout schemes [2](Line 16). The solution in Figure 2 has the following shortcomings and challenges.

- *C1.* The number of shares in the pool may be large, thus resulting in an unwieldy number of messages sent to the contract. For example, it may take $1,000,000$ shares on average to get a valid block. A naïve solution might require miners to create $1,000,000$ transactions and send all of them to the pool's contract. No existing open network agreement protocol can process that many transactions within the course of a few minutes [19, 20]. On the other hand, reducing the number of shares per block by increasing the share difficulty will increase the variance in reward for miners, thus negating the sole advantage of pooled mining (see [2] for more analysis on the effects of share difficulty).

- *C2.* A valid share earns miners a small amount of reward, but miners may have to pay much more in Ethereum gas fees when submitting their shares to the pool. The gas fee compensates for the storage and computation required to verify shares and update the contract state (see [21, 22]). Thus, `StrawmanPool` may render a negative income for miners when the fee paid to submit a share outweighs the reward earned by the share itself.

- *C3.* In Ethereum, transactions are in plaintext; thus, any network adversary can observe other miners' transactions that include the shares and either steal or resubmit the shares. This challenge does not exist in centralized pools where miners can establish secure and private connections to the pools. In decentralized settings, such secure connections are not immediate since i) there is no centralized operator who can initiate secure connections to miners, and ii) there is no PKI between miners in the pool. Thus, a good design for a mining pool must prevent the adversary from stealing others' shares. Similarly, the pool should prevent miners from over-claiming their contribution by either re-submitting previous shares or submitting invalid shares. Centralized pools can efficiently guarantee this since the pool manager can check every submission from miners.

- *C4.* This challenge is specific to the scenario when one wishes to use SmartPool for a different cryptocurrency (e.g. Bitcoin) than the one on which its contract is deployed (e.g. Ethereum). A smart contract in Ethereum running a Bitcoin mining pool must guarantee correct payments in Bitcoin. This is tricky because Bitcoin miners expect to receive rewards in Bitcoin, but Ethereum contracts can operate only on balances in Ether.

3 Design

SmartPool's design can be used to implement a decentralized mining pool on Ethereum for many existing target cryptocurrencies, but for ease of explanation we fix Ethereum as the target. In Section 5, we discuss how one can use SmartPool-based decentralized mining pools for other cryptocurrencies (e.g. Bitcoin).

3.1 Approach

We briefly describe how we address the challenges from Section 2.2 in SmartPool.

- SmartPool guarantees the *decentralization* property by implementing the pool as a smart contract. Like any smart contract, SmartPool is operated by all miners in the Ethereum network, yet it can secure other cryptocurrency networks including Bitcoin as well as the underlying Ethereum network itself. SmartPool relies on the Ethereum's consensus protocol to agree on the state of the pool. The security of SmartPool depends exclusively on the underlying network (*i.e.* Ethereum) which runs smart contracts, not on how many users adopt the pool.
- SmartPool's *efficiency* comes from allowing miners to claim their shares in batches, *e.g.* one transaction to the SmartPool contract can claim, say, 1 million shares. Furthermore, miners do not have to submit data of all shares but only a few for verification purposes, hence the transaction fee per share is negligible. As a result, the number of transactions required to send to SmartPool is several orders of magnitude less than the number of shares (*i.e.* the number of messages in P2Pool).

- We propose a simple but powerful probabilistic verification of submissions from miners. Our mechanism, aided by a new and efficient Merkle-tree based commitment scheme, guarantees the same average outcome as running a full verification for each submission by enforcing a *penalty* function to disincentivize cheating. Our mechanism detects miners submitting duplicated shares or resubmitting shares in different batched claims. As a result, we guarantee *fairness* in that miners receive their expected reward based on their contributions even when other dishonest miners submit invalid shares.
- SmartPool forces the miner to commit the right set of beneficiary addresses in the share before mining, so that it cannot be changed after a solution is found. This commitment prevents share theft, wherein a network participant tries to use someone else's solutions to pay itself.
- For the case of running an external SmartPool-based Bitcoin mining pool on top of Ethereum, SmartPool leverages the Bitcoin `coinbase` transaction to guarantee that miners can mine directly in their target currency (*i.e.* Bitcoin) without trusting a third party to proxy the payment (e.g. between Ethereum and Bitcoin). Nevertheless, miners still need to acquire Ether to pay for the gas when interacting with the SmartPool smart contract. Such costs are less than 1% of miners' reward as we show in our experiments with a deployment in Ethereum testnet. Indeed SmartPool operates at lower cost than today's centralized pools.

3.2 Overview of SmartPool

SmartPool is a smart contract which implements a decentralized mining pool for Ethereum and runs on the Ethereum network. SmartPool maintains two main lists in its contract state — a claim list `claimList` and a verified claim list `verClaimList`. When a miner submits a set of shares as claim for the current Ethereum block, it is added to the `claimList`. This step acts as a cryptographic commitment to the set of shares claimed to be found by the miner. Each claim specifies the number of shares the miner claims to have found, and it has a particular structure that aids verification in a subsequent step. SmartPool then proceeds to verify the validity of the claim, and once verified, it moves it to the `verClaimList`. Claim verification and payments for verified claims happen atomically in a single Ethereum transaction. Each claim allows miners to submit a batch of (say, 1 million) shares. Submitted claims need to include sufficient meta-data for verification purposes. During the first step of mining the shares, if a miner finds a valid block in the target cryptocurrency, it can directly submit the found block to the target cyrptocurrency network with the SmartPool address as the beneficiary. Thus, miners receive payouts for their shares one or more blocks after SmartPool receives reward from the target network; and, the mechanism ensures that the cryptographic commitment strictly preceeds the verification step (the cryptographic reveal phase).

Field Size (bytes)	Name	Data type
4	number	uint
32	parent hash	uint
32	TRIEHASH(TX_list)	uint
20	coinbase address	address
32	state root	uint
32	extra_data	char[32]
8	timestamp	uint
8	difficulty	uint
8	nonce	uint

Table 1: Some important fields of a block header in Ethereum. "coinbase address" is the address that receives the block reward, while "extra_data" allows miners to include any data (upto 32 byes) to the block header.

In Section 3.4 we will discuss our verification protocol, a key contribution of this work which enables efficiency. The goal of the verification process is to prevent miners from both submitting invalid shares and over-claiming the number of shares they have found. SmartPool pays claimants proportional to the number of shares claimed, if the verification succeeds, and otherwise nothing. The key guarantee here is that of fairness — SmartPool does not advantage miners who cheat by claiming invalid or duplicate shares. The expected payoff from cheating is the same (or worse) as honestly reporting shares.

In order to join the pool, miners only need to prepare a correct block template. SmartPool maintains the `verClaimList` array in the contract which records the contributed shares by different miners to date. To enable efficient verification checks, SmartPool forces miners to search for blocks with a particular structure and dictates a particular template for claim submissions, which we discuss in Section 3.3. Unlike P2Pool, SmartPool miners do not have to run an additional consensus protocol to agree on the list state.

3.3 Claim Submissions

Miners can submit a large batch of shares in a single claim. To permit this, SmartPool defines a `Claim` structure which consists of a few pieces of data. First, the miner cryptographically commits to the set of shares he is claiming. The cryptographic commitment goes via a specific data structure we call an augmented Merkle tree, as discussed in Section 3.5. The Merkle root of this data structure is a single cryptographic hash representing all the shares claimed and is included in the Claim as a field called ShareAugMT.

After a miner claims several shares in a batch, SMARTPOOL requires the miner to submit proofs to demonstrate that the shares included in the claim are valid. For each claimed share being examined, SMARTPOOL defines a ShareProof structure to help validate the share. First, SMARTPOOL requires a Merkle proof, denoted as AugMkProof, to attest that the share has been committed to ShareAugMT. Furthermore, SMARTPOOL ensures that if a miner finds a share that is a valid Ethereum block, then the corresponding block reward is distributed among the pool members. In an Ethereum block, there is a special field called "coinbase address" which specifies the address that receives the block reward. A share in SMARTPOOL is valid only if the miner uses pool's address as the "coinbase address."

It is straightforward to see how SMARTPOOL's use of cryptographic commitments prevents certain timing vulnerabilities. SMARTPOOL asks the miners to fix their coinbase address before starting to find shares. Once a share is found, it is not possible to change or eliminate the coinbase address. SMARTPOOL also asks miners to put their beneficiary address in the "extra_data" field, so SMARTPOOL can extract the address to credit the share to. Although miners may use different addresses to submit their claims to the contract, SMARTPOOL credits the share to only one account by fetching the beneficial address from the "extra_data" field. This prevents miners from claiming the same share to different Ethereum addresses (or accounts), forcing a one-to-one mapping between shares found and addresses credited for them. If a network attacker steals someone else's share, it cannot pay itself since the coinbase transaction has already committed to a payee.

3.4 Batching & Probabilistic Verification

SMARTPOOL processes share claims efficiently. Miners can claim multiple shares to SMARTPOOL in a single submission. Each Claim includes less than one hundred bytes consisting of a cryptographic commitment for the shares, in a field called ShareAugMT. This cryptographic commitment forces the miner to commit to a set of shares before including them in the claim. Ideally, before accepting any claim of n shares submitted by the miner, we want to verify that

(i) all shares submitted are valid;
(ii) no share is repeated twice in a claim;
(iii) each share appears in at most one claim.

Probabilistic verification. For efficiency, SMARTPOOL uses a simple but powerful observation: if we *probabilistically verify* the claims of a miner, and pay only if no cheating is detected, then expected payoffs of cheating miners are the same or less than those of honest miners. In effect, this observation reduces the effort of verifying millions of shares down to verifying one or two!

We provide a way to sample shares to verify, outline a detailed procedure for checking validity in Section 3.5, and a full proof in Section 4. Here, we explain this observation with an example, since it may appear counter-intuitive at first. Let us consider a case where cheating miner finds 500 valid shares but claims that he has found 1000 valid shares to SMARTPOOL. If SMARTPOOL were able to randomly sample one share from the miner's committed set, and verify its validity, then the odds of having detected the cheating is $500/1000$ (or $1/2$). If the miner is caught cheating, he is paid nothing; if he gets lucky without being detected, he gets rewarded for 1000 shares. Note that the expected payoff for such a miner is still 500, computed as $(0.5 \cdot 1000 + 0.5 \cdot 0) = 500$, which is the same as that of an honest miner that claimed the right amount of valid shares. The argument extends easily to varying amounts of cheating; if the cheater wishes to claim $1,500$ shares, he is detected with with probability $2/3$ and stands to get nothing. The higher his claim away from the true value of found shares, the lower is the chance of a successful payout. By sampling $k \geq 1$ times, SMARTPOOL can reduce the probability of a cheater remaining undetected exponentially small in k, as we show in Section 4.

Searching for shares. To enable probabilistic verification, SMARTPOOL prescribes a procedure for mining shares. Each SMARTPOOL miner is expected to search for shares in a monotonic order, starting from a distinct value that it commits to. Specifically, when a miner claims shares $S = \{s_1, s_2, \ldots, s_n\}$, SMARTPOOL extracts a unique counter from each share, e.g., taking the first k (say 20) bits, and requires that the counters of all $s_i \in S$ to be strictly increasing. Each time a miner finds a valid nonce that yields a valid share, he increases the counter by at least 1 and searches for the next share. When the miner claims for the set S, its submitted elements must be lexicographically ordered by counter values. The miner commits the latest counter in his Claim to this set S, which has at most one share for each counter value. This eliminates any repeats in claimed shares in one claim, and across claims by one miner. In SMARTPOOL implementation as an Ethereum contract, as discussed in Section 3.5, we use the share's timestamp and the used nonce to act as the counter value of a share.

SMARTPOOL guarantees that miners produce distinct shares by providing a unique value in the "extra_data" field in each miner's share template. This ensures that miners search in distinct sub-spaces of the search space.

Checking Validity of Shares. SMARTPOOL checks that miners have followed the prescribed mining procedure by randomly sampling a share from each submitted

Claim along with a ShareProof (as described in Section 3.3). SMARTPOOL validates the following:

(i) the hash of the share meets the difficulty criterion;
(ii) the share is constructed correctly, i.e., uses the SMARTPOOL's address as the beneficiary address of the block reward.
(iii) the share correctly satisfies the proof-of-work (PoW) solution constraints (e.g. the use of predetermined 1GB dataset mandated by the Ethereum PoW scheme)

The checks for (i) and (ii) are straightforward. The check for (iii) is to guarantee that miners actually have and use the data cache when they generate the shares. This 1GB of data cache is introduced in Ethereum to make its PoW ASIC-resistant. Thus, skipping checking (iii) would allow rational miners to easily mine a lot of invalid shares and still get paid from SMARTPOOL. It is not straightforward to efficiently check (iii) inside a smart contract. Indeed a naïve solution would require a massive amount of gas and hence invoke enormous transaction fees. We discuss implementation tricks on how to check (iii) in Section 6.1.1.

It remains to discuss (a) how miners cryptographically commit to a batched set of shares in a claim, (b) how SMARTPOOL verifies that the committed set has monotonically increasing counters, and (c) how shares are sampled. For (a) and (b), one can think of using a standard Merkle tree on all the claimed share set to generate the cryptographic commitment. However, in a standard Merkle tree, verifying the inclusion of a share is efficient, but checking the ordering of the set elements is not efficient. In SMARTPOOL, we devise a new data structure called *augmented Merkle tree* to help us verify inclusion and ordering of shares efficiently.

3.5 Detailed Constructions

In this section, we discuss an efficient verification scheme using probabilistic share sampling and a simple penalty function that penalizes cheaters. The description here takes an Ethereum pool as a target, but the same data structure works for other PoW-based cryptocurrency such as Bitcoin as we discuss in Section 5.

Augmented Merkle tree. Recall that a *Merkle tree* is a binary tree in which each node is the hash of the concatenation of its children nodes. In general, the leaves of a Merkle tree will collectively contain some data of interest, and the root is a single hash value which acts as a certificate commitment for the leaf values in the following sense. If one knows only the root of a Merkle tree and wants to confirm that some data x sits at one of the leaves, then holder of the original data can provide a "Merkle path" from the root to the leaf containing x together with the children of each node traversed in the Merkle tree. Such a path is difficult to fake because one needs to know the children's preimages for each hash in the path, so with high probability the data holder will supply a correct path if and only if x actually sits at one of the leaves.

For the purposes of submitting shares in SMARTPOOL, we not only want to ensure that shares exist in the batch list but also that there are no repeats and ordering of the counters is correct. We therefore introduce an augmented Merkle tree structure which we use to guard against duplicates in the leaves.

Definition 1 (Augmented Merkle tree)**.** Let ctr be a one-to-one function that maps shares to integers. An *augmented Merkle tree* for a set of objects $\mathsf{S} = \{s_1, s_2, \ldots, s_n\}$ is a tree whose nodes x have the form $(\mathsf{min}(x), \mathsf{hash}(x), \mathsf{max}(x))$ where:

(I) $\mathsf{min}(x)$ is the minimum of the children's min (or $ctr(s_i)$, if x is a leaf corresponding to the object s_i),
(II) $\mathsf{hash}(x)$ is the cryptographic hash of the concatenation of the children nodes (or $\mathsf{hash}(s_i)$ if x is a leaf corresponding to the object s_i), and
(III) $\mathsf{max}(x)$ is the maximum of the children's max (or $ctr(s_i)$, if x is a leaf corresponding to the object s_i).

An augmented Merkle tree is called *sorted* if all of its leaves occur in strictly increasing order from left to right with respect to its counter function.

SMARTPOOL expects claims of submitted shares to be monotonically ordered by their counters. Thus, one can think of each share s_i to have a "timestamp" given by its $ctr(x)$, since integer-valued counters can be naturally ordered (ascending or descending). For implementation in Ethereum, we can use the block timestamp and an the nonce to serve as the counter. In Appendix 10.2, we discuss alternative candidates for the ordering function ctr with backward compatibility to serve Bitcoin mining.

Figure 3 gives an example of an augmented Merkle tree based on four submitted shares with timestamps as $1,2,3,4$ respectively. To prove that the share c has been committed, a miner has to submit two nodes d and e to SMARTPOOL. SMARTPOOL can reconstruct other nodes on the path from c to the root (*i.e.* b and a sequentially) and accepts the proof if the computed root is the same as the committed one. The proof for one share, thus, in a Merkle tree of height h will contain h hashes. The algorithm to check the validity of a proof for a valid path in an augmented Merkle is in Algorithm 1.

Batch submission with augmented Merkle trees. After collecting a list of shares, the miner locally constructs an augmented Merkle tree for all the shares in the list. It then submits the data of the root node of the tree along with a number indicating how many shares it finds to

Algorithm 1 Algorithm to verify the validity of one path in a augmented Merkle tree

```
procedure ValidateNodeInPath(x)
Check if x is a leaf:
    if $isALeaf(x)$ then
        if $!(x.min == x.max == x.ctr)$ or $!isValidShare(x)$ then
            return false
        end if
    else
        $left \leftarrow x.leftChild$
        $right \leftarrow x.rightChild$
        if $!isHashValid(x, x.hash)$ then
            return false
        end if
        if $(!(x.min < x.max)$ or $!(left.min == x.min)$
            or $!(right.max == x.max)$
            or $!(left.max < right.min))$ then
            return false
        end if
    end if

Check if x is the root:
    if $isRoot(x)$ then
        return true
    else
        return $ValidateNodeInPath(x.parent)$
    end if
end procedure
```

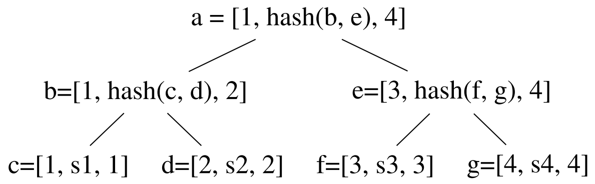

Figure 3: A sorted augmented Merkle tree for a list of shares (s1 to s4) with timestamp values from 1 to 4.

SmartPool. For example, the miner in Figure 3 submits the node *a* as the cryptographic commitment, which has min and max as 1 and 4 respectively. We use this committed data to i) verify that the sampled shares are found before the miner submits the claim; ii) efficiently check whether a share is duplicated in a claim. Verifying i) is straightforward as mentioned before. To verify ii), we observe that any duplicated shares in a claim will yield a sorting error in at least one path of the augmented Merkle tree. Thus, by sampling the tree in a constant number of leaves and checking their corresponding paths, with high probability we will detect a sorting error in the augmented Merkle tree if there is one.

Prevent over-claiming shares across claims. Our augmented Merkle tree allows us to detect if miners over claim shares or submit invalid shares in a claim. However, it does not help guarantee that miners do not submit the same shares in two different claims, *i.e.* overclaiming shares across claims. We prevent this problem by tracking counters of the shares in every claim and randomizing the counter start scheme for each claim. For example, we can use the the pair (block-timestamp, nonce) as a counter in an Ethereum block. We observe that, for a single miner, the counters for each claim are distinct because of the nonce. At the same time, timestamps monotonically order shares across claims, since the block timestamp monotonically increases over time.

Thus for any two distinct claims, the maximum share counter among an earlier claim is always smaller than the minimum counter of the shares in a later one. This observation enables a simple duplication check on the shares submitted in two different claims. Specifically, we require miners to submit their claims in chronologically increasing order of timestamp values (which are prefixes in the counter values). We use an additional variable last_max in our smart contract to keep track of the maximum counter (*i.e.* max value of the root node in the augmented Merkle tree) from the last claim. We only accept a new claim if the min value of the root node is greater than last_max, and update last_max properly if the new claim is valid.

Penalty scheme. Miners are rewarded according to the amount of shares that they submitted to the pool. In centralized pools, the pool manager is able to check every share submitted by miners, thus miners cannot cheat. In SmartPool, since we use probabilistic verification, we introduce a *penalty scheme* that penalizes detected cheating, independent of the reward distribution scheme used. The penalty scheme in Definition 2 is simple and suffices to disincentivize cheating, assuming rational miners.

Definition 2 (Penalty Scheme)**.** In SmartPool, the penalty scheme for a claim of *n* shares is as follow:

$$\begin{cases} \text{Pay all } n \text{ shares if invalid share was not detected;} \\ \text{Pay 0 otherwise.} \end{cases}$$

In Section 4, we prove that our penalty scheme disincentivizes rational miners from submitting wrong or duplicated shares. Our detailed analysis shows that for $k \geq 1$ samples, honesty maximizes payout.

Randomly sampling shares. In order to randomly sample, we need a source of randomness. A practical way to obtain such a random seed to use the hash of a future block. To reduce the amount of bias that any adversary can introduce to the block hash, one can take several samples based on several consecutive block hashes. For example, let us consider a scenario where a miner submits a claim of 1 million shares at block 1, and we wish to sample 2 random shares for our probabilistic verification. The miner is required to submit the data of 2 shares which are corresponding to hashes of blocks 1 and 2 (*e.g.* the hash values modulo 10^6) to SmartPool for verification. If the miner fails to submit any of these determined shares, they will not be able to claim the reward.

Putting everything together, we summarize the entire SmartPool protocol in Figure 4 of the Appendix. Due to the space constraints, we address other technical questions in the full version of the paper [23].

4 Analysis

We analyze the security that SmartPool provides through probabilistic verification and the penalty scheme in Definition 2.

We begin by informally reviewing the properties of our Merkle tree test and then formally establishing its correctness in Corollary 9 below. The intuition is if a claim has n valid shares and m invalid or duplicated ones, by randomly sampling a share from the claim, we can detect invalid shares with probability $m/(n+m)$. Suppose that a claim submitted by the adversary has n valid shares and m invalid or duplicated ones. If our test procedure is correct, the probability that our test on k independently chosen random samples fails to catch the cheating is at most $(1-\frac{m}{n+m})^k$. In this case, the cheating miner gets paid for $n+m$ shares, which is higher than reward for being honest (i.e. n shares). Corollary 9 shows that for all choices of m, for $k \geq 1$, the adversary's advantage (expected payoff) from cheating does not exceed the guaranteed payoff he would obtain from honestly submitting shares. Further, it is easy to see that over all choices of m the attacker's advantage is bounded by a negligible function in k (the number of samples checked).

Note that we establish that the adversary's advantage is minimal using a simple penalty function presented in Definition 2. Our probabilistic verification with penalties provide a basis to determine which shares to pay; however, any rewarding scheme can determine how to pay for the valid shares (e.g. using PPS, PPLNS, and so on).

Finally, we consider other possible attacker manipulations. One further security concern, in particular, merits analysis. The seed for our sampling is based on a block hash chosen by miners. We show that this source of randomness has a (low) bias, assuming that at least 50% of the mining network is honest. However, we establish in Theorem 10 that by sampling $k \geq 2$ times, the expected reward from honest submissions majorizes the expected payoff advantage from biased sampling.

4.1 Analysis of Expected Payoffs

We first analyze the scenario where the adversary cannot drop Ethereum blocks to introduce bias on sampling random seed, so the sample blocks in our probabilistic scheme are randomly selected. Furthermore, we assume that the adversary does not attempt to manipulate the expected format of the submitted data aside from possibly submitting duplicate or invalid shares. We will relax these conditions in Section 4.2.

It suffices for the SmartPool contract to check a single, randomly chosen path through a submitted augmented Merkle tree in order to pay fairly for shares, on average (Corollary 9). If all submitted shares are valid and there are no duplicates, then SmartPool pays for all shares with probability 1 (Theorem 7). The following facts will be useful.

Lemma 3. *For any node x in a augmented Merkle tree,*

(I) $\mathsf{min}(x)$ *is the minimum of all nodes below x, and*

(II) $\mathsf{max}(x)$ *is the maximum of all nodes below x.*

Proof. We will prove (I), and (II) follows by symmetry. Let y be any node below x, and trace a path from x to y in the given augmented Merkle tree. The min of x's immediate children along this path is, by definition of augmented Merkle tree, no greater than $\mathsf{min}(x)$. Similarly for the next children down, and so on, down to y. Therefore $\mathsf{min}(x) \leq y$. □

Proposition 4. *Let A be an augmented Merkle tree. The following are equivalent:*

(I) *A is sorted (see Definition 1).*

(II) *For every node x, the* max *of x's left child is less than the* min *of x's right child.*

Proof. We argue by induction. Assume (I), and further assume than (II) holds restricted to the first n levels above the leaves (the leaves are at the ground, *i.e.* zero level). Consider a node x at depth $n+1$. By the inductive hypothesis, the max of x's left child is less than the min of the next right child down, which is less than the min of the next right child down and so on, all the way down to some leaf y. By a symmetrical argument, the min of x's left child is greater than some leaf z which happens to be to the right of y. Since A is sorted, it follows that $\mathsf{min}(x) < y < z < \mathsf{max}(x)$.

Next assume (II), and let y and z be any two leaves. Let x be the lowest node (farthest from the root) which is an ancestor of both y and z. By Lemma 3, y is less than or equal to the max of x's left child, and z is is greater than or equal to the min of x's right child. Now $y < z$ follows from the assumption, hence A is sorted. □

Definition 5. A node in an augmented Merkle tree which satisfies condition (II) of Proposition 4 is called *valid*. Furthermore, we say that a path from a root to a leaf is *valid* if all its constituent nodes are valid. A path which is not valid is *invalid*.

The adversary can submit any arbitrary tree with the syntactic structure of an augmented Merkle tree, but not satisfying the constraint outlined in Definition 1. Let us call such a tree which syntactically has the structure of a augmented Merkle tree, but not necessarily satisfy the Definition 1 simply as a Merkle tree. A submitted Merkle tree can have any number of invalid or duplicate shares as well as ill-constructed internal nodes. Intuitively, an Merkle tree with invalid nodes will have *sorting errors*, which are defined below, and include both duplicates as well as decreasing share counters.

Definition 6. An element x in an array is *out of order* if there exists a corresponding witness, namely an element to the left of x which is greater than or equal to x, or an element to the right of x which is less than or equal to x. A leaf in a Merkle tree contains a *sorting error* if its label value is out of order when viewing the leaves' labels as an array.

Now, we will show that any submitted Merkle tree has at least as many invalid paths as the sorting errors it has.

Theorem 7. *Let A be a Merkle tree. If A is sorted, then all paths in A are valid. If A is not sorted, then every leaf containing a sorting error lies on an invalid path.*

Proof. If A is sorted then all its nodes are valid by Proposition 4, hence all paths in A are valid. Now suppose A is not sorted, and consider the highest node x in the tree (farthest from the root) which is is an ancestor of two distinct leaves y and z where y is left of z but $z \leq y$. Now x is not valid, because by Lemma 3 the max of x's left child is at least y and the min of x's right child is no more than z. It follows that neither the path from root to y nor the path from root to z is valid because both pass through x. □

The theorem above shows that miners who submit sorted augmented Merkle trees will receive their proper reward. Algorithm 1 checks the validity of a given path in a tree, and we omit a proof of its correctness here leaving it to inspection. It remains to demonstrate that sampling and checking a single path in the augmented Merkle tree suffices to discourage miners from submitting duplicate shares.

Corollary 8. *Every Merkle tree has at least as many invalid paths as sorting errors among the leaves. In particular, there are at least as many invalid paths as there are duplicate values among the leaves.*

Proof. Theorem 7 gives an injection from sorting errors to invalid paths. Since each duplicate and out of order leaf yields a sorting error, the result follows. □

Finally, we calculate the adversary's expected reward.

Corollary 9. *Under the payment scheme in Definition 2, if* SmartPool *checks one random path in the augmented Merkle tree of a claim, the expected reward when submit invalid or duplicated shares is the same as the expected reward when submit only valid shares.*

Proof. Suppose that in a claim of an adversary, there are k shares which are either invalid or duplicated. Since we randomly pick a path, by Corollary 8, we sample an invalid share with probability k/n and a valid share with probability $(n-k)/n$. Hence the expected profit from the payment scheme in Definition 2 is

$$\left(\frac{k}{n}\right)\cdot 0 + \left(\frac{n-k}{n}\right)\cdot n = n-k.$$

One expects to obtain this same profit by submitting only the $n-k$ valid shares. Thus, on average, it is not profitable to submit invalid shares to SmartPool if we employ the payment scheme in Definition 2 and check one random path from the augmented Merkle tree. □

In summary, SmartPool can efficiently probabilistically check that an augmented Merkle tree is sorted.

4.2 Discussion of Attacker Strategies

In this section, for clarity, we discuss ways in which an adversary might deviate from intended claim submission behavior and argue that these deviations do not obtain him greater rewards.

4.2.1 Rearrangements

The adversary cannot increase his expected profits by permuting the leaves of the Merkle tree. Observe that, given a list of integers L which may include repeats, a non-decreasing arrangement of L's members in the leaves of a Merkle tree minimizes sorting errors. By Theorem 7, every duplicate yields a sorting error regardless of permutation. Furthermore, the number of sorting errors that occur when the leaves are in non-decreasing order is exactly the number of duplicates. Hence a rational miner has no incentive to deviate from this non-decreasing configuration.

4.2.2 Bogus entries in augmented Merkle tree

Falsifying Merkle tree nodes does not decrease the number of invalid paths. Indeed, note that increasing the range for a given node can only increase the number of invalid paths, so we need only consider the case where the cheater makes the range smaller. If the range is made smaller so as to exclude the value of a leaf above that doesn't have a sorting error, then a new invalid path was introduced by cheating. If the range is made smaller so as to exclude a sorting error, then the path leading to that sorting error is still invalid, and therefore injection from Theorem 7 still applies.

4.3 Analysis of Bias In Seed Selection

We next consider the scenario in which the the adversary is able to drop Ethereum blocks to bias the random seed. Thus, the sample blocks in our probabilistic verification are not randomly selected, *i.e.* the adversary can drop the blocks which sample invalid shares from his claim. We show that, even in the extreme case where the adversary controls up to 50% of Ethereum mining power (*i.e.* can drop 50% of the blocks), it suffices to check only two randomly chosen paths through a submitted augmented Merkle tree in order to discourage the adversary from cheating.

Theorem 10. *If an adversary controls less than* 50% *of Ethereum hash power, then it suffices to sample only two paths of the augmented Merkle tree based on two consecutive blocks to pay miners fairly, on average.*

Proof. We call an Ethereum block a *good* block for the adversary if its hash samples a valid share in the adversary's claim. Suppose that in the adversary's claim, γ fraction of the shares are invalid ($0 \leq \gamma \leq 1$). By Theorem 7, at least γ fraction of the paths in the corresponding augmented Merkle tree are invalid. Hence, on average $1-\gamma$ fraction of the blocks are good blocks, since each block hash is a random number. The probability that the adversary's claim is still valid after two samples is the probability that two consecutive blocks in Ethereum are good blocks. We aim to compute this latter probability.

Let us assume that the choices of the two sample shares are drawn based on the hash of a single block hash, and that attacker controls p fraction of the network's mining power. The attacker's strategy is to successively drop blocks until he finds one that favorably samples his claim submission. We estimate his probability of success. The probability that he succeeds in exactly one round, regardless of who mined the block, is $(1-\gamma)^2$, that is, if the samples drawn are favorable. The chances that the attacker wins in exactly two rounds is the probability that the first block gave unfavorable sampling, but the attacker managed to mine it, and the next sample was favorable. The probability that all three of these independent events occur is $[1-(1-\gamma)^2] \cdot p \cdot (1-\gamma)^2$. In general, the chance that the attacker succeeds in exactly k rounds is

$$f(k) = \left(1-(1-\gamma)^2\right)^{k-1} \cdot p^{k-1} \cdot (1-\gamma)^2.$$

Summing over all possible game lengths k, we find that the chance that the attacker wins is exactly

$$\sum_{k=1}^{\infty} f(k) = (1-\gamma)^2 \cdot \sum_{k=0}^{\infty} \left[\left(1-(1-\gamma)^2\right) \cdot p\right]^k.$$

Since the right-hand side is a geometric series in which the magnitude of the common ratio is less than 1, we obtain

$$\sum_{k=1}^{\infty} f(k) = \frac{1}{1-(1-(1-\gamma)^2) \cdot p} = \frac{1}{1+(\gamma^2-2\gamma)p}.$$

The block withholding strategy is profitable if and only if this probability exceeds the attacker's chances of success without block withholding, namely $1-\gamma$. That is, the value p for which block withholding is advantageous satisfies

$$\frac{1}{1+(\gamma^2-2\gamma)p} > 1-\gamma. \tag{2}$$

We complete the analysis by inspecting the cases where p is greater than or less than the threshold $1/(2\gamma-\gamma^2)$. In the first case it follows that $p \geq 1/2$, since this threshold is always at least $1/2$ when $0 < \gamma \leq 1$, and if $\gamma = 0$ then the attacker has no incentive for dropping blocks. In the second case, the left hand side of (2) is negative, and so the inequality in (2) fails in this case. □

5 Supporting Other Cryptocurrencies

One can use SMARTPOOL's design to build decentralized mining pools for other cryptocurrencies. For clarity of exposition, we fix Bitcoin as the target in this section. The overall protocol is still similar to what have been discussed in previous sections, but here we present the detail changes to make SMARTPOOL work with Bitcoin while the contract is running on the Ethereum blockchain.

Generating a block template. In Ethereum, it is straightforward to to generate a valid block template, i.e., just by using the pool's address in the "coinbase address." It is tricker in Bitcoin since the block header is much simpler, (see Table 3 in Appendix 10.2) and the pool operates in another cryptocurrency (i.e., Ethereum). To generate a share that belongs to the pool, we leverage a special transaction in Bitcoin called a "coinbase transaction" whose outputs consist of a list of Bitcoin addresses paid and along with their payment amounts.

Specifically, in order to generate valid shares, a miner queries the `verClaimList` in the contract which records the contributed shares by different miners to date. The miner then prepares the coinbase transaction such that the first output pays to the miner who mined the block; the latter outputs pay to other miners included in the `verClaimList`. The sum of all outputs in the coinbase transaction equals the block reward. Thus, if a miner finds a fraction f of the shares in SMARTPOOL, he gets paid proportional to f in the reward that SMARTPOOL's miners get every time they mine a valid block.

Verifying a claim. As before, we use the probabilistic approach which samples random shares from a claim. However, in SMARTPOOL, verifying a Bitcoin share is slightly different from verifying an Ethereum share. Typically, a Bitcoin share is valid if the miner can demonstrate that the share has a valid coinbase transaction (labeled as the field `Coinbase`) in their `ShareProof` paid out to the pool members. The miner cannot selectively choose to omit this transaction; it is required to be the first transaction in the list of transactions (called `TxList`) on which the miner has searched for shares. The claimant must submit a Merkle root as commitment over the set `TxList` he has selected, and a Merkle proof (labeled `CoinProof`) that it contains the coinbase transaction. Second, the `ShareProof` contains an indication of the `verClaimList` based on which the payouts to miners were determined by the claimant. This last field is called a `Snapshot` to allow discretizing payouts over an ever-growing `verClaimList`. This is used to check the correctness of the `coinbase` transaction, i.e. if all the outputs pay to miners correctly. Figure 4 in the Appendix reports on all data fields of our Claim and ShareProof structures.

6 Implementation and Evaluation

We implemented SMARTPOOL and deployed it on Ethereum and Ethereum classic (main) networks. In this section, we describe the implementation (along with a Bitcoin pool implementation) and report actual fees from real mining that was done with SMARTPOOL.

6.1 Implementation

We implement SMARTPOOL protocol (as described in Figure 4) in an Ethereum smart contract and a miner software (client) that interacts with the contract according to our protocol [12]. Our smart contract implementation consists of two main modules, namely, *claim submission, claim verification.*

Claim submission. This module allows miners to submit their shares in batch. A miner submits a batch of shares by calling `submitClaim()` with the parameters: (i) the root of the corresponding augmented Merkle tree for the shares; (ii) number of shares in the tree; (iii) counter interval of the shares. A submission is accepted only if the smallest counter is greater than the current biggest counter.

Claim verification. A miner submits a proof for the validity of his last submitted claim by calling `verifyClaim()` with a branch in the augmented Merkle tree that corresponds to the next block hash. We allow different claims to include different amounts of shares, *i.e.* `NShare` can vary between claims. If the verification fails, then the claim is discarded, and the miner will not be able to submit all the shares (or a subset of them) again (forced by validating the counter in `submitClaim()`). If the verification is successful, then the claim is added to the to the `verClaimList` list.

6.1.1 Verifying Ethereum PoW

The PoW function that Ethereum is using is Ethash [16]. Ethash is not a native opcode nor a pre-compiled contract in the Ethereum virtual machine (EVM). Hence, to verify that a block header satisfies the required difficulty we have to explicitly implement Ethash function. Ethash was designed to be ASIC resistant, which is achieved by forcing miners to extract 64 values from pseudo-random positions of a 1 GB dataset. Thus, to explicitly compute Ethash, one would have to store 1 GB data in a contract, which costs roughly 33,554 Ether (storing 32 bytes of data costs 50,000 gas). Moreover, the Ethereum protocol dictates that the dataset is changed every four days (on average). Hence, one would require a budget of approximately $\$3,000,000$ per day as of June 2017 to maintain the dataset, which is impractical. Alternatively, one could store a smaller subset of the seed elements and calculate the values of the dataset on the fly. Unfortunately, to extract values from the seed one would have to compute several $SHA3\_512$ calculations, which is not a native opcode in the EVM, and would require massive gas usage if queried many times.

Fortunately, for our purposes, we do not need to fully compute Ethash. Instead it is enough to just verify the result of an Ethash computation. Thus, we ask the miner to submit along with every block header the 64 dataset values that are used when computing its Ethash and a *witness* for the correctness of the dataset elements. The *witness* shows that the 64 values are from the corresponding positions in the 1 GB dataset. Intuitively, to verify the witness for dataset elements, the contract will keep the Merkle-root of the dataset and a witness for a single element is its Merkle-branch. Formally, the pool contract holds the Merkle-roots of all the 1 GB datasets that are applicable for the next 10 years. We note that the content of the dataset only depends on block number (i.e., the length of the chain). Hence, it is predictable and the values of all future datasets are already known. Storing the Merkle roots of one year dataset requires storing 122 Merkle hashes, and would cost only 0.122 Ether.

We note that technically, our approach does not provide a mathematical guarantee for the correct computation of Ethash. Instead it guarantees the correct computation provided that the public dataset roots stored on the contract were correct. Hence, it is the miner's responsibility (and best interest) to verify the stored values on the contract before joining the pool. As the verification is purely algorithmic, no trust on the intentions of the contract authors is required.

6.1.2 Coinbase Transactions in Bitcoin

Recall that the payment to the Bitcoin miners is done via the coinbase transaction of a block. As per Figure 4, SmartPool allows miners to fetch the `verClaimList` and build the coinbase transaction locally. This approach, however, has a technical challenge regarding the transaction size when we implement SmartPool in the current Ethereum network. Specifically, a single coinbase transaction may have many outputs to pay to hundreds or thousands of miners. As a result, the size of the coinbase transaction could be in the order of 10KB (e.g., P2Pool's coinbase transactions is of size 10KB [24]). Hence, it is expensive to submit a coinbase transaction of that size to an Ethereum contract. In SmartPool implementation we could not ask miners to construct the coinbase transaction naively and submit as the input for `verifyClaim()` function.

To address the challenge, we modify SmartPool protocol slightly. Instead of asking miners to construct the coinbase transaction naively as in P2Pool, we ask them to work on only a small part of it. Specifically, we observe that we can fix the postfix of the coinbase transaction by using the pay per share scheme. Recall that the block reward consists of the block subsidy (12.5 Bitcoin) and the transaction fees. Thus, in our implementation, we pay the transaction fees to the miner who finds the block. The remaining 12.5 Bitcoin (the block subsidy) is paid to, say, the next 1 million shares in `verClaimList`. This distribution is encoded in all the latter outputs. Thus, we can fix all the outputs but the first one in the coinbase transaction, since the next 1 million shares in `verClaimList` are the same for all miners. This allows us to maintain the postfix of the coinbase transaction in SmartPool and only ask miners to submit the prefix (the first output) when they verify a share. Our approach significantly reduces both the gas fees paid for `verifyClaim()` and also the amount of bandwidth that miners have to send for verification.

Block submission. In SmartPool-based pool for Bitcoin, there exists the block submission module which allows any user to submit a witness for a new valid block in the Bitcoin blockchain so that SmartPool can have the latest state of the blockchain. If the block is mined by miners in SmartPool, SmartPool updates the `verClaimList` to remove the paid shares from the list. This also reduces the amount of persistent storage required in the contract since we do not need to store all verified claims in SmartPool.

Function	Gas	Price	% of reward
`submitClaim()`	79,903	0.000319612	0.01%
`verifyClaim()`	2,872,693	0.011490772	0.6%

Table 2: Ethereum fees of contract operations for Ethereum pool. Prices are in Ether. We note that in `verifyClaim()` for the Ethereum pool, 2.1M gas is spent on Ethash verification.

There are other technical subtleties in block submission and constructing coinbase transactions. We discuss these in Appendix 10.2.

6.2 Experimental Results

We deployed SmartPool on Ethereum [25] (and Ethereum classic [26]) live networks and mined with them with 30GH/s (4GH/s) hash power for 7 days (1 week). The pool successfully mined over 20 blocks [27] (85 blocks [28]) in corresponding periods. In this section we report the deployment cost of the contract and the fees that our protocol entails.

For `verifyClaim()`, we measure the cost to check 1 sample. The cost to check multiple samples can be easily computed from the cost to check 1. The results are presented in Tables 2.

The contract consists of over $1,300$ lines of Solidity code. The deployment of the contract consumed $4,351,573$ gas (6.24 USD). The contract source code is publicly available [29]. To reduce verification costs, we have submitted 1024 Merkle nodes for each 1GB dataset, namely, all the nodes in depth 10 of the Merkle tree. This operation was done is 11 transactions, which consumed in total around $6,000,000$ gas (around 15 USD) [30]. We emphasize that this operation is done only once every $30,000$ Ethereum blocks, or roughly 5 days. We report the evaluation of the claim submission and verification in transactions [31,32]. In our report, a miner with 20 GH/s submits a batch of shares every 3 hours. Every batch is rewarded with around 1.8 Ether (630 USD), and entails total gas fees of 0.011 Ether. Hence, the miner pays 0.6% for the effective pool fees.

7 Related Work

A number of previous works have studied the problem of addressing centralization in cryptocurrencies, and addressing flaws in pool mining protocols. We discuss these here, and further discuss security of smart contract applications of which SmartPool is an instance.

P2Pool. The work which most directly relates to SmartPool is P2Pool [8]. As discussed in Section 2.1, P2Pool consumes much more resources (both computation and network bandwidth), and the variance of reward is much higher than in centralized pools. SMARTPOOL solves these problems in P2POOL by i) relying on the smart contracts which are executed in a decentralized manner; ii) using probabilistic verification and a novel data structure to reduce verification costs significantly; iii) applying simple penalty scheme to discourage cheating miners. As a result, SMARTPOOL is the first decentralized pooled mining protocol which has low costs, guarantees low variance of reward to miners. Further, SMARTPOOL is more secure than P2POOL since any miner who has more than 50% of the mining power in P2POOL can fork and create a longer share-chain. On the other hand, the adversary has to compromise the Ethereum network to attack SMARTPOOL.

Pooled mining research. Several previous works have analysed the security of pooled mining in Bitcoin [2, 4, 9–11]. In previous works [9–11], researchers study the block withholding attack to mining pools and show that the attack is profitable when conducted properly. In [2] Rosenfeld *et al.* discussed (i) "pool hopping" in which miners hop across different pools to exploit a weakness of an old payoff scheme, and (ii) "lie in wait" attacks that allows miner to strategically calculate the time to submit his blocks. These challenges also apply to SMARTPOOL when SMARTPOOL is used as a decentralized mining pool in existing networks, and have specific payoff schemes to reward miners as solutions. The design of SMARTPOOL is agnostic to the payoff scheme used to reward miners. Furthermore, if SMARTPOOL were to be deployed natively in a cryptocurrency as the only mining pool (see Appendix 10.1), these attacks no longer work.

In [13], Miller *et al.* study different puzzles and protocols which either make pooled mining impossible and/ or disincentivize it. Out work is different from [13] in several aspects. First, we aim to provide an efficient and practical decentralized pooled mining protocol so miners have an option to move away from centralized mining pools. Second, SMARTPOOL is compatible with current Bitcoin and Ethereum networks as we do not require any changes in the design of these cryptocurrencies. In [13], the solutions are designed for new and future cryptocurrencies.

In [3, 4], the authors study the decentralization of the Bitcoin network. Previous works have highlighted that Bitcoin is not as decentralized as it was intended initially in terms of services, mining and protocol development [3, 33]. On the other hand, Bonneau *et al.* provided an excellent survey on Bitcoin which also covered the security concerns of pooled mining [4].

Smart contract applications. Previous works proposed several applications which leveraged smart contracts [34–36]. For example, in [35], Juels *et al.* study how smart contracts support criminal activities, *e.g.* money laundering, illicit marketplaces, and ransomware due to the anonymity and the elimination of trust in the platform. Such applications are built separately from the underlying consensus protocol of the network. In this work, we propose a new application of smart contract that enhances the security of the underlying network by supporting decentralized mining pools. Bugs in smart contract implementations are a practical concern; we belive the use of bug-detection tools such as Oyente [17] are useful to SMARTPOOL as well as other.

8 Conclusion

In this paper, we present a new protocol design for an efficient decentralized mining pool in existing cryptocurrencies. Our protocol, namel SMARTPOOL, resolves the centralized mining problem in Bitcoin and Ethereum by enabling a platform where mining is fully decentralized, yet miners still enjoy low variance in reward and better security. Our experiments on Ethereum and Ethereum Classic show that SMARTPOOL is efficient.

9 Acknowledgment

We thank Vitalik Buterin, Andrew Miller, Ratul Saha, Pralhad Deshpande, and anonymous reviewers for useful discussions and feedback on the early version of the paper. We thank 26 pseudonymous donors who donated to support the development of SMARTPOOL. We also thank Victor Tran and Andrew Nguyen for their work on the robust implementation of SMARTPOOL's client.

This work is supported by the Ministry of Education, Singapore under Grant No. R-252-000-560-112 and the European Research Council under the European Union's 7th Framework Programme (FP7/2007-2013, ERC grant no 278410). The deployment of SMARTPOOL on the main Ethereum network is supported by the Ethereum Foundation under a special development grant. All opinions expressed in this work are those of the authors.

References

[1] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *bitcoin.org*, 2009.

[2] Meni Rosenfeld. Analysis of Bitcoin pooled mining reward systems. *CoRR*, abs/1112.4980, 2011.

[3] Arthur Gervais, Ghassan O. Karame, Vedran Capkun, and Srdjan Capkun. Is bitcoin a decentralized currency? *IEEE Security and Privacy*, 12(3):54–60, 2014.

[4] Joseph Bonneau, Andrew Miller, Jeremy Clark, Arvind Naryanan, Joshua A. Kroll, and Edward W. Felten. SoK: Bitcoin and second-generation cryptocurrencies. In *IEEE Security and Privacy 2015*, May 2015.

[5] Maria Apostolaki, Aviv Zohar, and Laurent Vanbever. Hijacking bitcoin: Large-scale network attacks on cryptocurrencies. *To appear at IEEE Security and Privacy*, 2017.

[6] The problem of censorship. https://blog.ethereum.org/2015/06/06/the-problem-of-censorship/, June 2015.

[7] Bitcoin Wiki. getblocktemplate mining protocol. https://en.bitcoin.it/wiki/Getblocktemplate, November 2015.

[8] P2pool: Decentralized bitcoin mining pool. http://p2pool.org/.

[9] Nicolas T. Courtois and Lear Bahack. On subversive miner strategies and block withholding attack in Bitcoin digital currency. *CoRR*, abs/1402.1718, 2014.

[10] Ittay Eyal. The miner's dilemma. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, SP '15, pages 89–103, Washington, DC, USA, 2015. IEEE Computer Society.

[11] Loi Luu, Ratul Saha, Inian Parameshwaran, Prateek Saxena, and Aquinas Hobor. On power splitting games in distributed computation: The case of bitcoin pooled mining. In *IEEE 28th Computer Security Foundations Symposium, CSF 2015, Verona, Italy, 13-17 July, 2015*, pages 397–411, 2015.

[12] SmartPool team. SmartPool's github. https://github.com/smartpool.

[13] Andrew Miller, Ahmed Kosba, Jonathan Katz, and Elaine Shi. Nonoutsourceable scratch-off puzzles to discourage bitcoin mining coalitions. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 680–691, New York, NY, USA, 2015. ACM.

[14] Cynthia Dwork and Moni Naor. Pricing via processing or combatting junk mail. In *CRYPTO*, 1992.

[15] Adam Back. Hashcash - a denial of service counter-measure. Technical report, 2002.

[16] Ethereum Foundation. Ethash proof of work. https://github.com/ethereum/wiki/wiki/Ethash.

[17] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016.

[18] N. Szabo. The idea of smart contracts. http://szabo.best.vwh.net/smart_contracts_idea.html, 1997.

[19] Loi Luu, Viswesh Narayanan, Chaodong Zheng, Kunal Baweja, Seth Gilbert, and Prateek Saxena. A secure sharding protocol for open blockchains. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 17–30, New York, NY, USA, 2016. ACM.

[20] Kyle Croman, Christian Decker, Ittay Eyal, Adem Efe Gencer, Ari Juels, Ahmed Kosba, Andrew Miller, Prateek Saxena, Elaine Shi, Emin Gün Sirer, Dawn Song, and Roger Wattenhofer. *On Scaling Decentralized Blockchains*, pages 106–125. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016.

[21] Loi Luu, Jason Teutsch, Raghav Kulkarni, and Prateek Saxena. Demystifying incentives in the consensus computer. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 706–719, New York, NY, USA, 2015. ACM.

[22] Ethereum Foundation. Ethereum's white paper. https://github.com/ethereum/wiki/wiki/White-Paper, 2014.

[23] Loi Luu, Yaron Velner, Jason Teutsch, and Prateek Saxena. Smart pool : Practical decentralized pooled mining. Cryptology ePrint Archive, Report 2017/019, 2017. http://eprint.iacr.org/2017/019.

[24] P2Pool's coinbase transaction. https://tinyurl.com/zrp3dod.

[25] SmartPool contract on Ethereum. http://tinyurl.com/yankgher.

[26] SmartPool contract on Ethereum classic. http://tinyurl.com/yd2b2ry7.

[27] Blocks mined by SmartPool on Ethereum. http://tinyurl.com/ycs4tpzb.

[28] Blocks mined by SmartPool on Ethereum classic. http://tinyurl.com/ya8xf8xm.

[29] Source code of a SmartPool-based ethereum mining pool. http://tinyurl.com/yc22jdjs.

[30] SmartPool set 1GB dataset contract. http://tinyurl.com/y7qt2khq.

[31] Transaction that submits a claim to an ethereum pool. http://tinyurl.com/ya9vbc2k.

[32] Transaction that verifies a claim in an ethereum pool. http://tinyurl.com/y9an3pwp.

[33] Arthur Gervais, Ghassan Karame, Srdjan Capkun, and Vedran Capkun. Is bitcoin a decentralized currency? In *IEEE Security and Privacy*, 2014.

[34] Thedao smart contract. https://daohub.org/, 2016.

[35] Ari Juels, Ahmed Kosba, and Elaine Shi. The ring of gyges: Investigating the future of criminal smart contracts. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 283–295, New York, NY, USA, 2016. ACM.

[36] Jason Teutsch, Loi Luu, and Christian Reitwiessner. Truebit: A verification and storage solution for blockchains. https://medium.com/@chriseth/truebit-c8b6a129d580#.d2txm5usu, 2016.

10 Appendix

10.1 Applications

We discuss several applications that can be built based on SmartPool. One straightforward application is to build decentralized mining pools for cryptocurrencies as we have established. Apart from requiring low costs, guaranteeing low variance in rewards to miners than the only related solution P2Pool, SmartPool is also more secure. Specifically, one must compromise the entire Ethereum network (*e.g.* having more than 50% of Ethereum network) in order to compromise SmartPool. On the other hand, the adversary only needs to acquire 51% of P2Pool's mining power in order to build the longest share-chain in P2Pool and rule out other miners' contributions.

The second application is a new cryptocurrency based on SmartPool in which mining is fully decentralized. Typically, we enforce the consensus rules such that only blocks generated by SmartPool are accepted valid blocks. One can easily build a SmartPool-based cryptocurrency by using our introduced solution and adding the aforementioned consensus rule which dictates that only SmartPool can produce new valid blocks. Such cryptocurrencies can offer several good properties to the network that existing cryptocurrencies cannot. First, mining is fully decentralized, yet miners still enjoy low

Field Size (bytes)	Name	Data type
4	version	int32_t
32	prevBlock	char[32]
32	TxMerkleRoot	char[32]
4	timestamp	uint32_t
4	bits	uint32_t
4	nonce	uint32_t

Table 3: Header of a Bitcoin block. This is also used as the header for shares in pooled mining.

variance in reward. This improves the security of the underlying network as a whole significantly. Second, miners are not susceptible to several attacks targeting to pooled mining. For example, in [9–11] the authors demonstrate that if a malicious miner withholds blocks from a victim pool and mines privately in other pool, the miner can earn more profits from the loss of miners in the victim pool. Such block withholding attack does not work in SmartPool-based cryptocurrencies since there is only one pool in the network.

10.2 Implementation Subtleties for SmartPool-based Bitcoin pool

In this section we address two technical issues that arise from the design of the protocol. The first issue is the format of a witness for a new valid block, and the second issue is how a miner should decide on his coinbase transaction in the next share he mines.

Witness for a new valid block. Intuitively, a witness for a new block is a block header (see Table 3) with sufficient difficulty. However, in Bitcoin network (like in any blockchain based network), some of the mined blocks could be *orphan*, namely, they could be transmitted to the network a short period before or after an *uncle block* (a block that extends the a previous block but does not reach the blockchain) was found. In this case the network will eventually form a consensus over only one of the blocks, and the other block(s) will become orphan (and will not get any block reward from the network). In our protocol we must update the miners `verClaimList` list only according to non-orphan blocks. For this purpose, as a witness we ask for a chain of six blocks. While in theory, even a chain of six blocks could become orphan, in practice this never happens.

Deciding on the coinbase transaction of the next share. In order for a share to be valid it must have a coinbase transaction that corresponds to a `verClaimList` list. However, the `verClaimList` list is updated by the Ethereum contract. Hence, the contract is only aware of the Ethereum timestamp at the time the list is updated. On the other hand, the function `verifyClaim()` is supposed to verify the coinbase transaction according to the Bitcoin timestamp of the share. Hence SmartPool must synchronize Bitcoin and Ethereum time-stamps. The synchronization is done by introducing a new time metric, namely, the number of blocks SmartPool has found. With this new notion of timestamp, we implement the `verClaimList` list in such way that a list of payment claims is maintained for every block number n. The list of n corresponds to the payments that have to be done when SmartPool finds block number n. As new blocks might be reported with some delay, a payment request for a bulk that is verified in time n is added to the payment list of time $n+20$.

Given this implementation, the miner should construct the coinbase transaction in time n in the following way: As long as a new block is not found, the coinbase should correspond to list n. Once a new block is added to Bitcoin's blockchain, the miner should immediately start working on list $n+1$ (which already exists, as it was constructed at time $n-19$), even before the new block is submitted to the contract. If the new block becomes orphan, the miner should switch back to list n. Otherwise, after six blocks he should submit a witness for block n.

We note that in this approach the miner might do some stale unrewarded work in case the new block ends as an orphan block. However, such cases are also not rewarded in standard pools.

Other candidates for counter. Careful readers may realize that the timestamp field has only 4 bytes, thus we will run out of values for the counter after 2^{32} shares. In SmartPool, one can have several ways to implement the share's counter. For example, one can embed the counter inside the coinbase transaction of a share. Specifically, Bitcoin allows users to insert 40 random bytes in a transaction output after the `OP_RETURN` opcode [7]. SmartPool can force miners to store the share's counter in these 40 bytes, which can accommodate much more number of shares (*i.e.* 2^{320}).

[7] https://en.bitcoin.it/wiki/OP_RETURN

Notations

- Let `NSize`, `NSample` denote the number of shares included in a claim and the number of random samples SMARTPOOL will verify in each claim respectively.
- Let `claimList`$[x]$ store all unverified claims submitted by the miner at address x.
- Let `verClaimList`$[x][y]$ store all verified and unpaid claims submitted by the miner at address x at block y.
- Let `maxCounter`$[x]$ store the maximum counter of the miner at address x.
- We denote d as the minimum difficulty of a share.

Data structures.

The Claim structure has the following fields.

1. the number `NSize` of claimed shares;
2. the `ShareAugMT` commitment of the set of claimed shares.

The ShareProof structure for a share s_i has the following fields.

- the header of the share s_i located at the i-th leaf in the augmented Merkle tree;
- the `AugMkProof`, attesting that s_i is committed to the `ShareAugMT`;

For SMARTPOOL-based Bitcoin pool, the following additional data fields are included in the ShareProof

- the `Coinbase` transaction;
- the `CoinProof`, attesting that the coinbase transaction is included in the `TxList` of s_i; and
- the `Snapshot` of `verClaimList` that the `Coinbase` is computed on.

Main executions in SMARTPOOL

- **Accept a claim.** Accept a claim $\mathscr{C}$ which has the Claim structure and includes `NSize` shares from a miner x. Add $\mathscr{C}$ to `claimList`$[x]$ and update `maxCounter`$[x]$.
- **Verify a claim.** Receive a proof p which has ShareProof structure for a share s_i included in a claim $\mathscr{C}$ from miner x. SMARTPOOL verifies the following.

  1. if i is the supposed position that we want to sample based on the intended block hash;
  2. if s_i's hash is included in the claim $\mathscr{C}$ by verifying amkp_{s_i};
  3. if s_i meets the minimum difficulty d;
  4. if s_i's counter is greater than the last `maxCounter`$[x]$;
  5. if the coinbase address is the pool contract's address for Ethereum; or if `Coinbase` is included in s_i based on `CoinProof` and if `Coinbase` is correctly constructed with respect to `Snapshot` of `verClaimList` for Bitcoin.

  We reject the claim $\mathscr{C}$ if any of the above checks fail. If everything is correct and we have verified `NSample` from $\mathscr{C}$, update `verClaimList`$[x]$. Otherwise, wait for more proofs from miner x.
- **Get a new valid block** (for Bitcoin's pool only). If a new block is mined by SMARTPOOL, update `verClaimList`.
- **Request payment** (for Ethereum's pool only). When a miner requests his/ her payment, send the payment in proportional to his/her shares in `verClaimList`. Update `verClaimList` when the payment is done.

For miners

- **Construct block template.** For Ethereum, simply use the pool contract's address as the coinbase address. For Bitcoin, fetch `verClaimList` from SMARTPOOL and build the correct coinbase transaction locally.
- **Find valid shares.** Simply search for valid `nonce` which yields valid shares.
- **Submit a claim.** If have found enough `NSize` shares, build an augmented Merkle tree and submit a claim $\mathscr{C}$ to SMARTPOOL to claim these `NSize` shares.
- **Submit proofs.** Wait until $\mathscr{C}$ is accepted then construct and submit `NSample` proofs p_i ($i = 1, 2, \ldots, \texttt{NSample}$), each follows the ShareProof structure, to SMARTPOOL.

Figure 4: Summary of how SMARTPOOL protocol works for both the pool and miners.