Bypassing Non-Outsourceable Proof-of-Work Schemes Using Collateralized Smart Contracts

Alexander Chepurnoy[1,2], Amitabh Saxena[1]

[1] Ergo Platform
{kushti}@protonmail.ch, {amitabh123}@gmail.com
[2] IOHK Research
{alex.chepurnoy}@iohk.io

Abstract. Centralized pools and renting of mining power are considered as sources of possible censorship threats and even 51% attacks for decentralized cryptocurrencies. Non-outsourceable Proof-of-Work schemes have been proposed to tackle these issues. However, tenets in the folklore say that such schemes could potentially be bypassed by using escrow mechanisms. In this work, we propose a concrete example of such a mechanism which is using collateralized smart contracts. Our approach allows miners to bypass non-outsourceable Proof-of-Work schemes if the underlying blockchain platform supports smart contracts in a sufficiently advanced language. In particular, the language should allow access to the PoW solution. At a high level, our approach requires the miner to lock collateral covering the reward amount and protected by a smart contract that acts as an escrow. The smart contract has logic that allows the pool to collect the collateral as soon as the miner collects any block reward. We propose two variants of the approach depending on when the collateral is bound to the block solution. Using this, we show how to bypass previously proposed non-outsourceable Proof-of-Work schemes (with the notable exception for strong non-outsourceable schemes) and show how to build mining pools for such schemes.

1 Introduction

Security of Bitcoin and many other cryptocurrencies relies on so called Proof-of-Work (PoW) schemes (also known as scratch-off puzzles), which are mechanisms to reach fast consensus and guarantee immutability of the ledger. Security of such consensus mechanisms is based on the assumption that no single entity controls a large part of the mining power. For example, if a single entity controls 33% then it can earn unproportionally more rewards using *selfish mining* [1], and with more than 50% an adversary can do double spending or filter out certain transactions. However, individually, it is more beneficial for a miner to join a pool despite the fact that it is detrimental to the system as a whole since it causes concentration of mining power. Another threat, especially for new cryptocurrencies are potential Goldfinger attacks using hosted mining services to rent mining power in order to mine (or attack) a cryptocurrency [2]. Non-outsourceable scratch-off puzzles have

been proposed to address these issues [3,4], whose primary goal is to discourage pooled mining. Such approaches require reward spending to depend on some trapdoor information used in solution generation. A notable example of a real world implementation of this idea is Ergo [5], whose PoW, Autolykos [6], is based on [4]. In this work, we describe how to bypass the non-outsourceability of many such schemes, including Ergo. While our solution bypasses non-outsourceability, which gives the ability to form pools, we still retain a level of decentralization in our solution. In particular, our approach retains the *censorship resistance* property of non-outsourceable puzzles (see Section 2.2 for details.

The rest of the paper is organized as follows. Section 2 contains an overview of the current state of affairs in proof of work schemes and pooled mining along a high level overview of non-outsourceable puzzles. Section 3 describes one approach for creating mining pools in many types of non-outsourceable puzzles, specifically those of [4]. Section 4 describes another approach that covers a wider range of puzzles [4,3]. We conclude the paper in Section 5 along with pointers for future research.

2 Background

2.1 Proofs of Work

We first describe the vanilla PoW mechanism used in Bitcoin. A miner collects a number of unconfirmed transactions and builds a Merkle tree on top of them. The digest of this tree, denoted t here, is stored in a section of the block called the *block header*, which also includes the hash of the previous block's header h and a random string n called the nonce. We use the term m to denote the puzzle made of the concatenation of the Merkle tree digest and the hash of the previous block. That is, $m = t\|h$ and then the header is of the form $m\|n$. The solution is also determined by another parameter $\lambda > 1$, called the *difficulty*. Let H be a collision resistant hash function with output of 256 bits. The header $m\|n$ is considered a valid solution if $H(m\|n) \leq 2^{256}/\lambda$. A miner repeatedly tries different values of n (and possibly m) until a solution is found. Since H is like a random oracle, the probability of finding a solution in one attempt is $1/\lambda$. All PoW systems use the above idea of finding a value from a uniform distribution that falls within some narrower range based on the difficulty parameter.

2.2 Pooled Mining

Bitcoin allows mining pools, which roughly work as follows. The pool distributes work based on a some m that it decides. Each miner tries to find a solution for the given m and any solution found is sent to the network. A miner actually tries to find a *share*, which is like a solution but with reduced difficulty (also decided by the pool). Some of the shares may also be real solutions, which result in valid blocks. A miner gets paid by the number of shares submitted. This is possible because the Bitcoin PoW puzzle is a *scratch-off puzzle* [3], a type of PoW puzzle

that can be processed in parallel by multiple non-communicating entities with an appropriate reduction in search time.

The pool generates the potential block candidates as if it was solo mining, and then distributes that candidate to its miners for solving, which can be considered workers for the pool. The shares have no actual value and are just an accounting tool used by the pool to keep track of the work done by each worker. The key observation with pools is that miners do work for some other entity who then distributes the rewards back to the workers. Since the pool selects the transactions that are to be included, this gives the pool greater control over the entire blockchain network. We define this using three levels of (de)centralization that a pool may operate at.

1. *Level 1 (Centralized):* The pool operator defines both m and the reward address. Thus, a pool operator has full control over which transactions are included (censorship) and also carries the risk of losing the rewards.
2. *Level 2 (Censorship Resistant):* The pool operator does not define m but collects the rewards. This is resistant to censorship but still carries the risk of losing the rewards.
3. *Level 3 (Decentralized):* There is no centralized pool operator but rather another decentralized oracle that emulates the pool operator and rewards are automatically given to the participants based on the shares they submitted (see P2Pool [7] for Bitcoin and SmartPool [8] for Ethereum). In P2Pool, this oracle is implemented using another blockchain, while in SmartPool, it is implemented using a smart contract.

The following table summarizes the concepts.

Pool level	Censorship	Reward theft risk	Example
L1 (Centralized)	Yes	Yes	BTC.com
L2 (Censorship Resistant)	No	Yes	ErgoPool (this work)
L3 (Decentralized)	No	No	SmartPool[8], P2Pool [7]

The primary issue with pools is that they increase the potential of transaction censorship and 51 percent attacks. One way to address this issue is to disallow pools entirely. This is what non-outsourceable puzzles aim to achieve, and Ergo is the first practical implementation of such puzzles [5]. Thus, such puzzles are designed to provide the same level of security as an L3 pool.

We, however, note that disallowing pools entirely comes with its own set of problems. For instance, at Ergo launch, the difficulty went up so quickly that miners with single GPUs could not find any blocks in a reasonable time. Since Ergo does not allow pools, such miners had no incentive to continue mining. In fact, this research was motivated from the need to create a mining pool for Ergo. However, we also want our solution to retain the security offered by lack of pools, that is, resistance to censorship and 51% attacks.

Our solution is based on the observation that another way to address censorship and 51 percent attacks is to have pools operate at levels L2 or L3, where

these issues are not present. Thus, not only can we have decentralization in mining but also have all the benefits of pools (such as regular income for miners and thereby, stronger network). Our solution is designed for L2 but can also be trivially extended to operate at L1. Additionally, it may be possible to extend it to L3 using approaches similar to SmartPool or P2Pool, which we leave as a topic for further research.

2.3 Non-Outsourceable Puzzles

We start with overviewing (non-)outsourceability definitions in existing literature expressed in different works, such as Non-outsourceable Scratch-Off Puzzles [3], 2-Phase Proof-of-Work (2P-PoW) [9], PieceWork [4], Autolykos [6]. The details of these approaches are described in Sections 3 and 4. However, at a high level, all these approaches can be broadly classified into two categories.

In the first one [6,4,9], which we call **Type 1**, a PoW scheme is considered non-outsourceable if it is not possible to iterate over the solution space without knowing some trapdoor information (such as a secret key) corresponding to some public information (such as a public key) contained in the block header, with block rewards locked by that trapdoor information. The reasoning here is that in order to send the reward to a pool's address, each miner must know the secret corresponding to that address. However, a pool does not trust miners and so will not give the secret away to them.

In the other category [3], called **Type 2**, a PoW scheme is considered non-outsourceable if for any solved block, a miner can generate another block efficiently with non-negligible probability. The motivation behind this definition is that a miner can get paid for shares by trying to generate a block that pays the reward to the pool. In case of successful block generation, however, the miner could generate and broadcast another block that sends the reward to the miner instead of the pool. We further classify Type 2 into *weak* if the identity of the miner stealing the rewards can be ascertained and *strong* if the identity remains secret.

At a technical level, both Type 1 and 2 approaches rely on a miner's ability to steal the pool's rewards. The difference lies in the way this occurs. In Type 1 schemes, the miner is able to steal the reward *after* the block gets finalized. In Type 2, the reward can only be stolen *before* a block is finalized into the blockchain.

We note that all Type 2 schemes have an inherent problem that allows malicious actors to flood the network with a large number of valid but distinct solutions, thereby causing network partitions and instability. This causes the network to converge very slowly or result in several forks. Hence, we don't consider Type 2 schemes to be robust in reaching consensus, thereby making them impractical in the real world. We call this the *forking attack*. Strong Type 2 schemes are even more prone to this attack because there is no fear of detection.

In this work, we bypass the non-outsourceability of all Type 1 and weak Type 2 schemes assuming that their platforms support some minimal smart contract capability. The following table summarizes this.

Puzzle type	Thief's identity	When rewards stolen	Forking attack	Bypassed
1	revealed	after block acceptance	no	yes
2 (weak)	revealed	before block acceptance	yes	yes
2 (strong)	secret	before block acceptance	yes	no

2.4 Execution Context in Smart Contracts

To give understanding of how a smart contract can bypass non-outsourceability, we first explain what kind of data the contract can access.

In PoW currencies, a block contains a compact section called the *header*, which is enough to verify the PoW solution and check integrity of other sections (such as block transactions).

Execution context is what is available to a contract during execution. Considering UTXO-based cryptocurrencies, such as Bitcoin and Ergo, we can think about following components of the execution context. At the bare minimum, the first level, the smart contract should have access to the contents of the UTXO it is locking (i.e., its monetary value and any other data stored in it). At the second level, the smart contract may additionally have access to the spending transaction, that is, all its inputs and outputs. At the third level, the smart contract may have access to block header data in addition to the data at the second level. For example, in Ergo, the last ten block headers and also some parts of the next block header (which are known in advance before the next block is mined) are also available in the execution context. Finally, at the fourth level, the execution context may contain the entire block with all sibling transactions. Note that since the execution context must fit into random-access memory of commodity hardware, accessing the full blockchain is not a realistic scenario. The following table summarizes possible execution context components.

Context level	UTXO	Transaction	Header	Block	Example
C1	Yes	No	No	No	Bitcoin [10]
C2	Yes	Yes	No	No	–
C3	Yes	Yes	Yes	No	Ergo [5]
C4	Yes	Yes	Yes	Yes	–

3 Pooled Mining in Type 1 Puzzles

In a nutshell, Type 1 puzzles use a combination of two approaches: (1) The first approach is to replace the hash function with a digital signature (i.e., use public-key cryptography instead of symmetric key cryptography for obtaining the final solution) and (2) The second approach is to tie the public key to the rewards.

3.1 Using Public-Key Cryptography

The method requires a randomized signature scheme that is strongly unforgeable against adaptive chosen message attacks (s-UFCMA) and outputs signatures

uniformly spread over some range irrespective of how the signer behaves. Schnorr signature is one such scheme [11].

A candidate block header is constructed using transactions as in Bitcoin along with a public key p. A valid block header is a candidate block header along with a signature d that (1) verifies with this public key and (2) satisfies the difficulty constraints as before (i.e., is less than a certain value). The difficulty parameter is automatically adjusted as in Bitcoin.

One real-world implementation of this concept is Autolykos [6], the PoW algorithm of Ergo [5]. Autolykos uses a variation of Schnorr signatures [11], where the goal of a miner is to output d such that $d < 2^{256}/\lambda$ and λ is the difficulty parameter. The value d is to be computed as follows. First compute $r = H(m\|n\|p\|w)$ where m is the transactions digest, n is a nonce, p is a public key (an elliptic curve group element) and w is an ephemeral public key that should never be reused in two different blocks. Let x be the corresponding private key of w. Compute $d = xr - s$, where s is the private key corresponding to p.

3.2 Tying Public-Key to Rewards

The second technique in making a Type 1 pool-resistant scheme is to tie the rewards to the public key p contained in the block solution. That is, the platform enforces that any mining rewards are protected by the statement *prove knowledge of secret key corresponding to the public key p (from the block solution)*

We consider Ergo as an example here. Rather than enforcing this logic within the protocol, Ergo uses smart contracts to enforce it. In particular, this rule is enforced in a so called **Emission box**[3], a UTXO which contains all the ergs (Ergo's primary token) that will ever be emitted in rewards. The box is protected by a script that enforces certain conditions on how the rewards must be collected. In particular, it requires that a reward transaction has exactly two outputs, such that the first is another emission box containing the remaining ergs and the second is a box with the miners reward protected with the following script: *prove knowledge of the discrete logarithm (to some fixed base g) of group element p AND height is greater than or equal to the box-creation height plus 720.* This is possible because Ergo's (level C3) context includes the block solution.

The above approach ensures that the private key used for finding the block solution is also needed for spending the rewards. Consequently, anyone who finds a block also has the ability to spend those rewards. If we try to create any standard type of pool, we find that anyone having the ability to find a solution also has the ability to spend (i.e., steal) the reward. In fact, any standard pool must share the same private key among all participants, thereby making it impossible to determine the actual spender. This restriction also applies to decentralized schemes such as P2Pool and SmartPool because they both require that rewards be sent to addresses not under the miner's control.

[3] A box is just a fancy name for a UTXO. We will use these two terms interchangeably.

3.3 Creating a Mining Pool

We now describe a pooling strategy for bypassing any Type 1 scheme, provided that the underlying smart contract language supports context level C3 or higher (see Section 2.4). Hence one way to mitigate our method would be to restrict the smart contract language to level C2 or lower. Our concrete implementation uses Ergo as the underlying platform, which supports C3 context.

We will follow the *pay-per-share* approach, where the reward is distributed among the miners based on the number of shares they submitted since the last payout. Our pool is designed to operate at centralization level L2, where the pool only collects the rewards but does not select transactions (see Section 2.2). Hence, it provides resistance against censorship and does not encourage 51% attacks that are possible at L1. Note that the pool could also operate at L1 by creating miner-specific blocks using pair-wise shared public keys. However, this increases computational load on the pool and overall network usage, thereby reducing efficiency.

Basic variant: We first describe a basic version that is insecure, and thereby does not work in practice. We then incrementally enhance this version to patch the vulnerability to obtain the full version.

The key observation in our approach is that in a valid share, the reward need not necessarily be sent directly to the pool's address. What is actually necessary is that an amount equivalent to the reward is sent to the pool's address. This simple observation allows us to create a pool with the following rules:

1. Each miner can send the reward to his own public key p, whose secret key only he knows (*reward transaction*).
2. The block must also have another transaction sending the same amount as the reward to the pool address (*pool transaction*).

A valid share is a solution to a block with the above structure. A miner can efficiently prove that a share is valid without having to send the entire block to the pool. It can simply send the pool transaction along with the Merkle proof that validates that the transaction [12]. A pool operator collects such shares (along with the proofs) and any funds thus received when a block is solved are distributed among the miners using the pay-per-share algorithm. To ensure that miners generate valid blocks, the pool randomly asks miners to provide full blocks corresponding to some of their shares and penalize those who cannot.

One drawback of this is that each miner must have sufficient collateral to cover the reward amount at any time, even though the reward becomes spendable only after a 'cooling-off period' (720 blocks in Ergo). Thus, there is a minimum period during which the collateral is spent but the reward is locked and cannot be used as further collateral. Therefore, for uninterrupted mining, each miner must keep the reserved amount of at least 2 rewards (possibly more depending on the expected time to find a block).

To overcome this drawback, a pool may provide incentives such as allowing the miner to keep a fraction of the reward (example for the current reward of 67.5 ergs in Ergo, the pool may require only 65 ergs to be sent to it).

The broadcast attack: Let Alice be a miner with public key `alice`. If such a system is used in, say Bitcoin, then the system becomes insecure. Once the pool-paying transaction is publicized, anyone (not necessarily Alice) may broadcast it (possibly by using it as their own pool transaction).

Enhanced variant: The enhanced protocol mitigates the above attack. This is possible because ErgoScript allows us to use the block solution in the context, using which we can secure the pool transaction as follows. Instead of paying to the pool from an arbitrary box (or boxes), Alice will instead store this collateral in a special box protected by the following script:

```
minerPubKey == alice
```

A box with this script does not require a signature because the above statement only fixes the miner's public key to `alice` and does not enforce any other spending condition. Thus, anyone can create a transaction spending this box. However the transaction is valid only if the block that includes it is mined by Alice. This ensures that the box can only be spent if and when Alice mines a block. Alice creates her pool transaction using this box as input and submits her shares and proofs to the pool as before. She need not even use a private channel for this purpose and can broadcast this publicly. This enables the possibility of L3 decentralization level that requires public shares [7,8] (see Section 2.2).

The above variant prevents the broadcast attack because knowing the pool transaction does not help the attacker in any way (since anyone can create that transaction without Alice's help). An attacker might try to spend Alice's collateral in a transaction paying to some address other than the pool address. However, Alice will discard such transactions when creating a candidate block and only include her pool paying transaction that spends the collateral. In the worst case, if Alice does not check for others spending her collateral, the mined block will still include her own pool-paying transaction double-spending the same collateral, thereby making the entire block invalid.

Full variant: Observe that the above collateral box is not spendable until Alice actually mines a block. Depending on her hardware and the global hash rate, this may take a very long time, and her funds will be stuck till then. We would like Alice to be able to withdraw her collateral at any time she decides to stop participating in the pool. This can be done as follows. Alice first sets another public key `aliceWithdraw` that she will use to withdraw the collateral (it is possible to keep `aliceWithdraw` = `alice`). The modified script is:

```
(minerPubKey == alice) || aliceWithdraw
```

The first condition, `minerPubKey == alice`, ensures that when used to fund the pool output, the miner must be Alice as in the enhanced variant. The second condition, `bob`, ensures that the coins are not stuck till Alice finds a block, because it allows Alice may withdraw the collateral at any time. Alice should fund the pool transaction by satisfying only the first condition and never the second condition, otherwise the broadcast attack becomes possible. The second condition is be used only for withdrawing collateral.

Note that the above allows everyone to create a transaction spending Alice's collateral box as long as Alice mines the transaction. Alice may have more than one collateral box protected by identical scripts. Thus, an attacker may try to spend Alice's box that is not used in the pool funding transaction. Of course, Alice should not include such transactions in her block. This requires Alice to implement additional checks. An easier solution is for Alice to use another public key, `aliceLock`, as below to ensure that only she can create a valid transaction.

```
((minerPubKey == alice) && aliceLock) || aliceWithdraw
```

The above broadcast attack mitigation strategy requires C3 context level (i.e., access to `minerPubKey`) and will not work in lower levels. One may envisage a hiding strategy at C2 context level, where the pool transaction is not revealed in a share (only a commitment is revealed). The actual transaction is revealed only if a block is found or when a miner later proves to the pool that the shares were correct. However, this is also insecure. First note that there are two types of broadcast attacks. The first is the *leak-from-share* attack. The second is the *leak-from-orphaned-block* attack, where the transaction is extracted from a mined block that ends up getting orphaned. The hiding strategy mitigates the first attack but not the second.

Weak Broadcast security: We can obtain a weaker form of broadcast security for C2 context level by assuming a trusted pool as follows. A pool-paying transaction is created as before by spending some arbitrary input and paying to the pool address. The miner sends the shares along with the proofs to the pool over a private channel. The pool is trusted not to misuse the transaction. This addresses the leak-from-share attack. To address the leak-from-orphaned-block attack, the following strategy is used. Assume that the box funding the pool transaction contains a unique identifier of Alice (such as her public key) and a script that enforces any spending transaction to pay the pool. Lets us call this *Alice's funding box*. The pool then enforces the following rules internally.

1. Any transaction it receives from Alice's funding box that was not mined by Alice is considered *irregular*.
2. Any irregular transaction using Alice's funding box is not considered for pool reward disbursement and the funds are refunded back to Alice.

It is possible for everyone to verify that a given pool transaction is irregular if everyone knows Alice's public key. Thus, a pool cannot deny the existence of an irregular transaction. Refunds can also be made verifiable in many ways, such as by requiring the pool to create another funding box for Alice, which can be publicly verified. We can additionally require that the new funding box be created in a transaction that consumes the irregular transaction's output.

4 Pooled Mining with Type 2 Puzzles

In Type 2 puzzles, a miner can produce (with non-negligible probability) an alternative block for the same PoW solution [3]. For concreteness, we will use

public key cryptography to illustrate this, as we did for Type 1 puzzles. However, our approach will work for any other implementation of such puzzles.

Recall that a Type 1 puzzle comprises of two steps: (1) embedding a public key p in the block header, whose private key is needed in generating the solution, and (2) tying the block rewards to p. A Type 2 puzzle can be considered a variation of a Type 1 puzzle, where Step 1 remains the same but Step 2 is modified so that the block rewards are not tied to p but instead to another public key a that is certified by p. In other words, the complete solution is defined using a tuple $(p, a, \mathsf{cert}_p(a))$, where $\mathsf{cert}_p(a)$ is a signature on a that verifies with p.

The rationale behind non-outsourceability is that a cheating miner knowing the private key of p can steal the reward as follows. When claiming shares, the miner behaves correctly. That is, it constructs the block so that rewards go to the pool public key a. However, if a real solution is found, the rewards are sent to the miner public a' by creating a certificate $\mathsf{cert}_p(a')$. Thus, as in Type 1 puzzles, the pool risks losing rewards if it shares secrets with miners.

Watermarking: In the basic Type 2 scheme, a pool can make it possible to identify stolen rewards by publicly fixing a watermark identifying its blocks in advance [3]. A watermark in this context is something that is preserved even if the pool key a is replaced by the miner key a'. A few examples are the certifying key p or, say, half the bits of the nonce. If such a watermark is used then it becomes possible to identify the cases when the block rewards are stolen.

Strong Type 2 puzzles: In the above design, it is possible to determine when the rewards are stolen. For instance, using the public key p as a watermark, a pool may declare in advance that for a given p, it only considers the pair (p, a) as valid and any other pair (p, a') indicates a theft. The stronger variant of Type 2 puzzles replaces signatures with zero knowledge proofs so that the two cases (block rewards stolen or not) become indistinguishable. Any Type 2 puzzle that is not strong is called *weak*.

We describe a smart contract that bypasses both Type 1 and (weak) Type 2 schemes. For sake of brevity, however, we only describe the Type 2 solution here. Recall that for such schemes, it is possible to detect when a particular watermark is being used in the block. In our approach, this watermark is attached to the miner instead of the pool. Thus, the pool with share pair-wise watermarks with every miner. Similar to the previous approach, we will also require the miner to lock some collateral that can be used by the pool to cover any rewards taken by the miner. We also require the smart contract language to make available in the execution context the block solutions for not only the current block header but also the last L block header prior to the current one.

Then a weak Type 2 scheme can be bypassed as follows. In order to participate in the pool, Alice creates an unspent box that locks collateral with the guard script: *payable to pool public key if at least one of the last L headers contains the watermarked solution.* The same solution will also work for Type 1 schemes there because the block header cannot be efficiently altered without also altering the embedded public key. In ErgoScript, for example, this can be implemented as: `poolPubKey && lastHeaders.exists(_.minerPubKey == alice)`.

The method `exists` of `lastHeaders` takes as input another method, say f, that takes as input an header and outputs a `Boolean`. The method f is applied to every element of `lastHeaders` and the output of `exists` is the OR of the outputs of f. In this case, f outputs true if the miner public key in the header is Alice's public key.

A miner is permitted to send the reward to any chosen address, since as soon as a valid block is generated, the collateral becomes spendable by the pool. One way the miner can try to save the collateral is to generate L blocks after the one with the watermark, but this case is unlikely for a pool user if L is big enough. In Ergo, for example, $L = 10$, and the chance to generate 11 consecutive blocks is very small [10].

Note that the above script locks the collateral until Alice find a block, which may never happen. Hence, as in the Type 1 case, we need to allow Alice to withdraw collateral if she desires. However, the solution used in Type 1 (i.e., simply appending '`|| aliceWithdraw`') will not work here because the pool does not immediately get the collateral when Alice gets the reward, but rather after at most L blocks. If we allow Alice to withdraw the collateral at any time, the she can withdraw it in the same block as the reward. One solution would be to allow Alice to withdraw the collateral only after some fixed height H, while her participation in the pool using this collateral ends at height $H - L$, after which she must use new collateral. For simplicity, we skip this deadline condition for withdrawing the collateral by the miner in case a block is not found for a long time. However, a real world implementation must consider this.

5 Conclusion and Further Work

Non-outsourceable puzzles have been proposed as a possible workaround for attacks that arise due to pool formation in PoW blockchains. Such solutions fall into two broad categories: Type 1, where the reward is directly bound to some trapdoor information used for generating the block solution (and thus, that information is needed while spending), and Type 2, where the reward is indirectly bound to the trapdoor information via a certificate. Type 2 schemes can be further classified into weak, where the identity of the miner is revealed, and strong, where the identity remains hidden.

In this paper we proposed two approaches to bypass non-outsourceability of Type 1 and weak Type 2 schemes to create mining pools, thereby 'breaking' them. Our pools operates at level L2 (censorship resistance), where the pool does not control transactions to be included in blocks but only collects the rewards (see Section 2.2). Such pools do not pose stability threats that L1 level pools do. Although our pools are most efficient when operating at L2, they can operate at L1 simply by having the pool create miner-specific blocks using their public keys. Note that both L1 and L2 carry the risk of funds loss due to operator compromise. A topic of further investigation is to have the pools operate at L3, where there is no risk of losing funds.

Only strong Type 2 schemes (where a miner does not provide a block solution in the clear, but rather provides an encrypted solution along with zero-knowledge proof of its correctness) remain unbroken. However, it should also be noted that strong schemes are not very practical as they require a generic zero-knowledge proof system which imposes heavy burden on both the prover and verifier. Thus, such schemes currently have no implementations in the real world. Additionally, we note that Type 2 schemes in their entirety have an inherent weakness that make them impractical for real world use: the high possibility of forking attacks.

Both our approaches rely on smart contracts acting as decentralized escrows and require the underlying programming language to allow predicates at context level C3 or higher (i.e., access to the block solution; see Section 2.4). Thus, one way to invalidate our methods would be to restrict the language context to level C2 or lower. Note that even level C2 contracts allow sophisticated applications such as non-interactive mixing, rock-paper-scissors, and even an ICO [13].

Another open issue in mining pools is that of block withholding [14], where the miner tries to attack the pool by submitting valid shares but discarding actual solutions. The need for collateral in our schemes may possibly affect the attacker's strategy. This will be considered in a follow-up work.

References

1. Ittay Eyal and Emin Gün Sirer. Majority is not enough: Bitcoin mining is vulnerable. *CoRR*, abs/1311.0243, 2013.
2. Paul D. McNelis. We need a financial james bond to prevent a goldfinger attack on bitcoin. `https://www.americamagazine.org/politics-society/2017/10/31/we-need-financial-james-bond-prevent-goldfinger-attack-bitcoin`, 2017.
3. 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*, pages 680–691. ACM, 2015.
4. Philip Daian, Ittay Eyal, Ari Juels, and Emin Gün Sirer. (short paper) piecework: Generalized outsourcing control for proofs of work. In *International Conference on Financial Cryptography and Data Security*, pages 182–190. Springer, 2017.
5. Ergo Developers. Ergo: A resilient platform for contractual money. `https://ergoplatform.org/docs/whitepaper.pdf`, 2019.
6. Autolykos: The ergo platform pow puzzle. `https://docs.ergoplatform.com/ErgoPow.pdf`, 03 2019.
7. Xavier Chesterman. *THE P2POOL MINING POOL*. PhD thesis, Ghent University, 2018.
8. Loi Luu, Yaron Velner, Jason Teutsch, and Prateek Saxena. Smartpool: Practical decentralized pooled mining. In *26th {USENIX} Security Symposium ({USENIX} Security 17)*, pages 1409–1426, 2017.
9. How to disincentivize large bitcoin mining pools. `http://hackingdistributed.com/2014/06/18/how-to-disincentivize-large-bitcoin-mining-pools/`, 06 2014.
10. Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. `https://bitcoin.org/bitcoin.pdf`, 2008.

11. Claus-Peter Schnorr. Efficient signature generation by smart cards. *Journal of Cryptology*, 4(3):161–174, 1991.
12. Mike Hearn and Matt Corallo. Bitcoin improvement proposal 0037. Web document, October 2012.
13. Advanced ergoscript tutorial. `https://docs.ergoplatform.com/sigmastate_protocols.pdf`, 03 2019.
14. Nicolas T Courtois and Lear Bahack. On subversive miner strategies and block withholding attack in bitcoin digital currency. *arXiv preprint arXiv:1402.1718*, 2014.