A New Approach For Bitcoin Pool-Hopping Detection

Eugenio Cortesi[a,b], Francesco Bruschi[b], Stefano Secci[a] and Sami Taktak[a]

[a]Conservatoire national des arts et métiers, Paris, France.
[b]Politecnico di Milano, Milan, Italy.

ARTICLE INFO

Keywords:
Bitcoin
Blockchain
Cryptocurrency
Consensus
Miners identification
Pool-hopping

ABSTRACT

Mining pool hopping is a phenomenon taking place in cryptocurrency networks such that a miner changes of pool overtime, in a regular or recurrent way, to increase its gains related to mining work rewards by the mining pools. This phenomenon is not well understood, also because of the lack of a precise pool-hoppers detection solution. In this paper, we propose a methodology for detecting the pool-hopping behavior in the Bitcoin network; we propose a deterministic framework exploiting the different time windowing phases (rounds, epochs) involved in the rewarding process. Our methodology includes a new algorithm to identify the miners, and a new algorithm to trace the revenue stream distribution. We assess the performance of our approach for the five mining pools with the highest hash rates during two three-month period in 2020 and 2021: we show that the phenomenon is still advantageous in terms of overall gains for the pool-hoppers. We also assess the fairness in stake: the phenomenon was known to be unfair in the beginning of the Bitcoin network due to the simple rewarding methods in place at that time, with single rewards higher for pool-hoppers than for miners; we show that this is no longer true and that the new rewarding policies now guarantee that pool-hopping is fair with respect to miners that do not perform pool-hopping. Nonetheless, we also show that the cumulative gain over time of pool-hoppers can be higher by 33% on median than static miners.

1. Introduction

Bitcoin is a decentralized digital currency whose use has been growing significantly in recent years. It is considered the first cryptocurrency created and, despite not being a legally recognized currency everywhere, it has triggered the launch of hundreds of other alternative cryptocurrencies. Its adoption by large companies as Tesla has recently further enlarged its spread and adoption, as well as the development of the whole cryptocurrency ecosystem.

Bitcoin uses cryptography to ensure the veracity of information, while being transparently accessible to everyone. It involves resource-intensive computational activity that becomes increasingly difficult and is used to validate new blocks for inclusion in the blockchain. It is performed by *miners*, who support the system by providing the computing resource they have available. To prove that they have allocated sufficient computational power to the system, they must provide a proof, i.e., the solution of a hashing problem, as a result of which the miner receives a reward. Given the stochastic characteristic of Bitcoin, the average reward for the single mining is very high but the probability of winning the validation is extremely low.

In this regard, the main components that orchestrate the computational resources of the miners are the mining pools. They emerged to provide a more stable and predictable income for miners by offering the ability to share computing resources. Mining pools play an essential role in the security and performance aspects of the network and, nowadays, dominate Bitcoin mining activity as they contribute to $\approx 99\%$ of the total hash rate [22]. Pools compete with each other to attract more miners in order to get a higher market share and have a better chance of getting a reward for mining. For these reasons, since the early years of Bitcoin, pools have applied a process of rationalization of mining operations to provide a solid foundation for the stability of the ecosystem [23] and to build miner retention policies.

One of the phenomena that pools aim to prevent is precisely pool-hopping, i.e., strategic behavior influenced by the attractiveness of mining in terms of expected payoffs increased at the expense of static miners. As the name recall, the consist in the practice of jumping between pools dynamically. Over the years, many attempts [3] [22] [18] have been made to detect pool-hopping, a particularly complex practice given both the anonymity of Bitcoin and the characterization of reward patterns within pools. Early pools implemented proportional reward methods in which hopping was highly profitable for hoppers and unfair to normal miners. With such a method, if every miner hopped regularly, it would lead to the standstill of all proportional pools, thus being a dangerous practice for the system [19]. Today, almost all pools no longer use proportional reward method in favor of new methods created specifically to fully mitigate pool-hopping [19]. Despite their original purpose, Meni Rosenfeld in [19] showed that fully mitigating pool-hopping is not possible. In fact, miners are free to leave and change pools at will, following any system change that makes one pool more favorable than another one. The benefit of these methods turns out to be more specifically related to making pool-hopping not unfair and preventing pool stalling. Furthermore, since it is inconvenient for pools to undergo frequent hash rate changes, these methods also aim to keep miners within a pool, making hopping non-trivial to perform.

In this paper we go beyond a preliminary study described in [3] on the detection of pool-hopping miners in the Bitcoin system, using it as a starting playground to define a more accurate algorithmic approach taking more realistic assumptions. Our key innovation is a deterministic sorting of the time windowing (rounds, epochs) involved in the rewarding

ORCID(s):

process and related transactions linkage to miners, hence allowing us to spot more miners than with the lower bound set that can be found with [3]. We propose a four-step methodology to empirically evaluate the hopping phenomenon [3]:

1. extraction of the revenue stream from the mining pools and assignment the author pool of each reward based on a coin ownership methodology;
2. unique identification of the miners through the use and adaptation of known heuristics;
3. placement of a miner in space and time accordingly to the payoffs received, but reformulating prior work in [3] (going beyond the distribution of payoffs at the conclusion of each round);
4. comparison of the work intervals of each miner to detect if cross-pooling is performed. To this end, we introduce a characterization of miner work based on epochs of multiple rounds.

We apply this 4-step method to the pools with the largest computing power, i.e., 5 pools covering 60% of the total power in 2020 and 2021. Our goal is to precisely measure the extent to which pool-hopping still occurs. In addition, we qualify which type of hopping is most common, whether it is simple pool switching or concurrent work. In particular, we want to ensure that there are no unbalanced gains between traditional miners and hoppers, as opposed to what has occurred with the use of the proportional method [19], so that we can ascertain the harmlessness of the phenomenon. For the sake of reproducibility, we open source the code in [6].

Two main empirical findings derive from our contribution. We report the experimental evidence that:

- the pool-hoppers still obtain higher rewards in the long-run, even upon the introduction of new rewarding rules within mining pools that were meant to discourage pool-hopping ;
- the single rewarding transactions do not have higher amounts than transactions of static miners, differently than with old rewarding rules.

The paper is organized as follows. Section 2 gives the background. Section 3 synthetically describes the four-step pool-hopping detection framework. Section 4 presents the revenue stream analysis (Step 1). Section 5 describes the miners identification problem (Step 2). Section 6 presents the approach to handle the analysis across multiple rounds (Step 3). Section 7 presents the hoppers detection method (Step 4). Section 8 describes the results. Section 10 concludes the paper. The diagram in Figure 1 show the key steps characterizing our detection strategy.

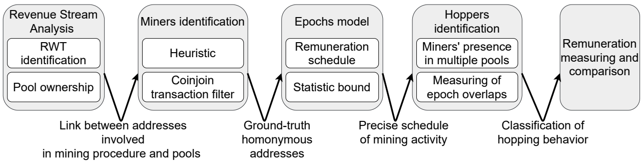

Revenue Stream Analysis
RWT identification
Pool ownership
Miners identification
Heuristic
Coinjoin transaction filter
Epochs model
Remuneration schedule
Statistic bound
Hoppers identification
Miners' presence in multiple pools
Measuring of epoch overlaps
Remuneration measuring and comparison
Link between addresses involved in mining procedure and pools
Ground-truth homonymous addresses
Precise schedule of mining activity
Classification of hopping behavior

Figure 1: Diagram representing the keys steps of our detection strategy.

2. Background

Bitcoin is a digital cryptocurrency, introduced in [15] and released in 2009. Instead of leveraging on physical entities, such as banknotes or real assets, Bitcoin relies entirely on data. It is a P2P distributed ledger technology (DLT), structured as a chain of blocks, the 'blockchain'. The blockchain data structure [14] links each validated block to the previous one by saving the hash identifier of the previous block in the new one. The first block of the chain is the 'genesis block'. A fundamental ingredient of blockchain is cryptography, offering secure data transmission and immutability. The transaction history is public and saved on a 'Ledger', that is a type of digital registry that contains transactions between parties specifically about business activities. The ledger allows for append-only operations, and everything stored in the past is immutable to protect the data from any kind of tampering and forgery. Bitcoin is distributed because it resides on multiple computing devices in different geographical locations. A computer node connected to Bitcoin supports the network by maintaining a copy of the ledger. Lightweight nodes connect to the system and download only the block headers, not the entire ledger. Bitcoin is a decentralized system, as data sharing and storage on the P2P network is allowed without entrusting control to any central authority; Moreover, its blockchain is permissionless, i.e., participation is public and open access without identification and that there is no need for a trusted third party as intermediary [1].

The operation of Bitcoin involves handling a large number of transactions, informing the network that the owner of a certain number of bitcoins has authorized the transfer of some of those to another user and ensuring the correctness with the asymmetrical cryptography [16]. Each Bitcoin transaction has one or more inputs and one or more outputs. Inputs correspond to the outputs of the transaction from which they derive, in the sense that the entire value of the previous transaction must be transferred with the new transaction and cannot be kept steady in the account. This is because Bitcoin uses the 'Unspent Transaction Outputs' (UTXO) model [2]. In case a user does not want to spend the entire value received, the system creates a transaction with two different outputs: one that transfers the desired value and one that transfers the difference to the sender. The transaction structure is designed to keep track of all the transactions financing a given one; it associates each input address with the hash of the transaction with which that amount was sent: this is the 'ID' field in the representation of a chain of transactions in Figure 2. Keeping the link between an input address and the position where it is located among the output addresses in the previous transaction is necessary to know what amount of bitcoins was supplied; for each input address, the position 'n' says where its amount is located among the output addresses in the previous transaction.

The data on the ledger must be the same for everyone to

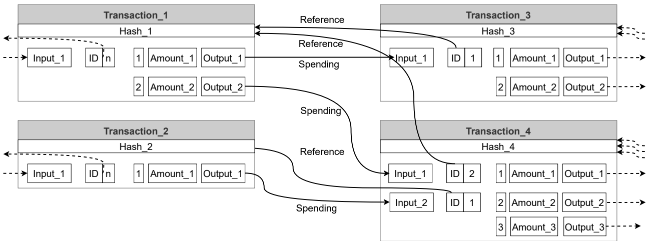


Figure 2: Chain of transactions.

prevent malicious actions and the process of reaching agreement among network participants on its correctness is called consensus. Bitcoin uses a ‘proof-of-work’ (PoW) consensus, the security of which is guaranteed by the fact that the majority of the hash power in the whole system will aim at extending the legitimate chain faster than any corrupt minority aiming at double spending [9]. The idea behind PoW is to make validation tasks difficult to perform, but easy to verify. Block validation consists of a competition between particular nodes, called ‘miners’, whose winner earns a reward. To be winner in the validation, a miner must solve a hashing problem. The nature of the problem relates the mining procedure to a lottery competition in which the validation success is completely random and the probability (p) of finding a valid hash within within time (t) is inversely proportional to the network difficulty and proportional to the energy spent. The ‘network difficulty’ (D) is the parameter that Bitcoin uses to keep the average time between blocks steady as the network’s hash power changes. The target difficulty of the problem is adjusted on the overall network difficulty every 2016 blocks and the adjustment is calculated comparing the actual blocks number added to the chain to the excepted one and adjust the new goal based on the varying percentage [27]. The hashing activity to solve the problem is identified as the process of finding the output of the SHA256 function given an input (S). This is a cryptographic hash function that maps strings S of arbitrary length to data strings of fixed length (h) [16]. S is composed of the hashed header of the block and a value, called ‘nonce’. To validate a block, it is necessary to find the nonce such that S in input to the SHA256 function generates an output starting with a fixed number of 0s (zeros). The number of initial 0s is set by the target difficulty, so that a full solution is found every 10 minutes. This prevents more blocks to be generated before the previous one’s successful propagation in the whole blockchain network. Eventually, when the nonce is found, the block is validated and attached to the blockchain. Both the nonce and the target difficulty are saved, so that the content of the block cannot be changed, otherwise the output of the hash function with the found nonce would not satisfy the target difficulty anymore. Bitcoin stores the nonce in the nonce field that is part of a particular transaction at the top of the block [26], the ‘coinbase transaction’: it is the one sending the block reward (B) to the miner for the completed validation and it has no input amount as it rewards with newly generated coins. The PoW mechanism ensures consistency on a probabilistic form [10], since forks can occur on the chain, namely two valid chains with the same block number. This inconsistent situation is resolved by validating a new block, as the chain with the highest number of blocks is considered the valid one. Transactions reach the confirmation only when are included in the longest chain.

To reduce remuneration uncertainty and variability over time, miners may choose to join a *mining pool*, that is a cooperative scheme in which multiple miners share their efforts to validate blocks. A pool has a hash rate variable in time, that depends on the number of participants sharing their resources on a certain moment. Rewards to miners are divided among them based on their contribution, allowing to earn a fraction of rewards on a regular basis. The working period it takes to a pool to validate a block is called ‘round’. The coinbase transaction includes a string inserted by the miner or the pool completing the round, in order to keep record of who is the responsible of the validation won and, therefore, deserves to receive the block reward. To show that the miners are dedicating their computing power, the pool manager asks them to provide partial solutions that satisfy a lower target difficulty than the original one. Specifically, any partial solution whose output begins with 32 zeros, which number is fixed, is called ‘share’. The more computing power dedicated to the problem, the more shares a miner finds on average. Depending on the rewarding strategy of the pool, the operator could retain a fee (f), calculated as a percentage of the reward for a single share. Mining pools set a specific difficulty (d) to set a lower bound for shares. A miner, which has submitted a certain number of shares to contribute to the validation process, is rewarded through what are called ‘rewarding transactions’. Sometimes, the miner is also awarded with the fees paid by users who submit transactions. The fee is an incentive for the miner to include the transaction in their block.

2.1. Miners population estimation

Some estimations guess that around 1 million miners exist [11]. Are those all active and unique miners? The answer is no. The number comprises both miners that contributed to validations in the past and now remain inactive nodes and the ones that subscribed to more pools simultaneously, resulting in different identities that should be considered unique. Moreover, in a given pool a miner can submit shares using different addresses, resembling again several distinct identities. Given these aspects, the conditional tense is compulsory when guessing the miners population.

Another estimation poses the attention on the number of nodes that constitute the blockchain; e.g., in May 2019 Bitcoin reached 100K full nodes, which would lead one to say that a realistic estimation of miners would be at least lower than that. Roughly half of full nodes in the blockchain are listening nodes - ordinary users - so a guess could be that half of them are not mining. Moreover, one could assume an even lower number knowing that almost 60% of full nodes run vulnerable code and just 30% of the blockchain full nodes are up to date [4].

Actually, the number of miners can be also higher than the number of nodes. Nowadays, the majority of pools host a full node on their server, avoiding miners to become themselves a full node and making them participate in the validation remotely through it. Some pools that offer the possibility to be part of the pool without downloading a complete replica, act as aggregators and offer a service for which the miner remains extraneous to the validation process as it joins the pool only at the application level. These miners do not actually become part of the network, do not own a wallet of bitcoins and do not even have assigned public keys, as they will be rewarded by the pool outside of the Bitcoin system. They, therefore, remain completely invisible to those who want to analyze the data maintained by the blockchain, but on the other hand it would not make sense to study the pool-hopping phenomenon for this kind of locked-in miners. In the analysis of the blockchain, one is limited to consider miners that we can call *backbone miners*, i.e., present in the ledger and for which we see one, or many, identifiers.

2.2. Rewarding methods

Within a mining pool, the most frequently adopted rewarding methods are the following.

Proportional: the method is the first one adopted by miners pools, and it is known to have been used mainly in the early stages of the Bitcoin system. Participants are rewarded proportionally to the number of shares they submit compared to the total amount of shares needed for a given validation. The reward per share is calculated as the block reward divided by the number of shares in the round, hence a share submitted early in the round will have a higher reward on average than a share submitted later, making *early mining* more profitable than continuous mining.

Slush: the method is built on the proportional one, but it assigns a score to shares. The block reward at round end is distributed among participants in proportion to their score. The score given for each share depends on the amount of time that has elapsed since the round started. The more time has passed, the higher the score.

Pay-Per-Share (PPS): an instant reward is assigned to the miner for each completed share; it is fixed and pre-defined in advance, so whether the pool successfully mines the block or not, miners get rewarded for every share submitted anyway [20]. This method is riskier with higher variance in the expected duration of rounds: if the the process lasts longer, the pool has to pay for every share submitted, so it could go bankrupt. Transaction fees go to the pool.

Full-Pay-Per-Share (FPPS): very similar to ordinary PPS, with the only difference that the pool also distributes transaction fees if the block is validated.

Pay-Per-Last-N-Share (PPLNS): the miner gets the payout proportionally to the shares it submits, but only when a certain amount of shares (N) is reached. N is set as a function of the difficulty (D) and it can be up to double D (rounded down to integer) [3]. It is fixed regardless of boundaries of rounds [5] and it does not depend on luck, so whether the blocks end up in the chain of not, each share increases N. If the pool fails to mine a block, the miners do not receive rewards for the shares submitted for that validation. Only the shares that contributed to the validation of blocks eventually in the chain are rewarded. The reward is calculated proportionally to a coefficient applied to all submitted shares, which varies from 0 to 1 as the number of submissions increases. At 1 the miner is using its maximum computational power. For example, if a pool estimates that 1000 shares are required for the calculation, the estimated power to receive a reward will gradually increase from 0 to a decimal of the maximum hash rate as they submit shares. After the first 1000 shares are received the pool starts rewarding the miner in full and only for shares that contributed to the validation of blocks that get in the chain.

Pay-Per-Share-Plus (PPS+): the miner's reward is paid out on the expected value similar to PPS. The Transaction fees are paid out on a PPLNS method, meaning that they are distributed to miners based on how much hash rate they contributed.

2.3. Pool-Hopping phenomenon

The pool-hopping phenomenon is a behaviour that depends on the attractiveness of mining in terms of expected payoffs. Some pools may be more or less attractive in the sense that they may reward miners more in some circumstances and less in others. Pool-hopping consists of mining for a pool only when its attractiveness is high, and leaving it when the attractiveness is low. In [19] it is shown that in particular situations pool-hopping turns out to be more profitable than continuous mining, and that at the expense of other miners. At first it was thought that, in order to comply with the principle of cooperation and consortium of miners, all miners should contribute equally to the pool in both good and bad times and their reward should average out statistically. Nonetheless, pool-hopping is not unfair in all scenarios, as not always it compromises the rewarding of the work of other miners or the whole system. Indeed, it is in the freedom of a miner to change pools to seek more suitable conditions. This behavior is not desirable for pools that undergo continuous changes in hashing power; in order to discourage pool-hopping, pools use specific rewarding mechanisms and follow miners retention policies. The practice of pool-hopping cannot be completely prevented, as there are several factors external to the rewarding dynamics that encourage miners to engage in it [19].

The proportional method is the less protected against pool-hopping and no longer used in most cases. It was particularly suitable for the phenomena because, as explained in 2.2, *early mining* was very profit table. Meni Rosenfeld [19] proved that, with the proportion method, as long as the number of shares in the round is lower than 43.5% of the difficulty, a submitted share has a higher reward. The negative consequence is that honest miners that participate in the pool continuously, could theoretically and in the worst case receive 43.5% less. It was found that if everyone hops, all proportional pool will eventually come to a standstill as miners would reach a number of shares in the current round equal to 43.5% of the difficulty, at which point no-one will mine there anymore [19]. The gain that can be achieved by following this strategy is up to 28.1% [13], and is at the expense of continuous miners, depending on the ratio between their hash rate versus the hoppers' one.

Slush's method [19] is one of the first designed to combat pool-hopping. It prevents the phenomenon on round length basis, but it is an incomplete solution, as the attractiveness of pools is ultimately affected by fluctuations in hash rate over time.

Modern methods such as PPS, PPLNS are widely used and were implemented with the original purpose of completely mitigate pool-hopping. Authors in [8] confirm that PPLNS strongly decreases pool-hopping altogether, as the more mining you do, the more shares you earn, stimulating miners to mine continuously and over the long term. On the contrary, as discussed in [19, 24], there are factors for which plain PPLNS cannot be considered a definitive solution and other versions need to be considered. From a theoretical standpoint and to a lesser extent than with proportional method, the phenomenon still has reasons to be carried out when adjustments of difficulty (D) and block reward (B) occur, as some pools would be more attractive, influencing the strategic actions of the miners. In addition, with this method a miner could decide to send shares simultaneously to different pools because, although N would be reached more slowly, it would smooth out the probability of working on validating a block that eventually does not end up in the blockchain, thus receiving no reward.

Pools implementing PPS, PPS+ and FPPS also are claimed to be hopping-proof, as the miners' payout is predetermined and is a percentage of the actual hash rate they provide. Actually, the opposite is true because each pool independently determines the fixed cost of each share based on network complexity, network rewards, blocking time, and pool power. In addition, because shares for blocks that do not end up in the chain are still rewarded, the pool manager compensates for its risk by increasing the mining fee, creating an imbalance, once in favor of the miners, once in favor of the pool. With both methods the phenomenon is not likely to occur multiple times on a round length basis and is less likely to be disadvantageous to counterparts, but still they leave space to dynamically switch pools between rounds.

2.4. Related work

The phenomenon of pool-hopping has been a subject of research almost since the beginning of Bitcoin. As we introduced, over time the phenomenon has evolved, and the assumptions surrounding it have changed. There are several works that have theoretically studied the phenomenon as a function of the rewarding method adopted and the resulting gain [19]. Others have attempted to designate a strategy for measuring it and understanding its ground-truth. The detection has also varied as some dynamics have been described in greater depth.

Authors in [18] presented reward payout flow patterns of 3 pools: BTC.com, AntPool, and ViaBTC over 4 weeks. They describe the dynamics used by the pool to redistribute rewards to miners, consecutive to receiving the block reward. What deserves particular attention is that the distribution of rewards to participants is not immediate. Instead, the pools follow transaction patterns, whereby the amount of bitcoins controlled by the pools themselves is managed in order to succeed in reward transactions in a phased manner. The patterns are similar for some pools, while changing frequently for others.

Authors in. [22] also empirically studied the phenomenon of pool-hopping across the ledger to identify and define pool migrations in Bitcoin history. They focused on how miner pools evolved to adapt to the behaviors of miners moving between them, creating a system of competition. Their algorithm for extracting payment streams from miner pools is similar to the one we propose next, with one main difference regarding the association of pool ownership to the transaction. Regarding the identification of miners, in contrast to [3], they claim that the heuristic of [17] is not accurate enough, as it neglects the existence of consequences of mixing services. Therefore, they use a mathematical model and a known entity dataset for the association.

Our work is the follow up of the preliminary study in [3] on empirical pool-hopping detection. They investigated pool hopping between KanoPool and SlushPool during April 6–20, 2016; the starting point is the miners identification, solved via a classification, sorting, and matching of all coinbase, rewarding, and ordinary transactions. Starting from each receiving address of a rewarding transaction, the presented method searches for the address among all the sender addresses of ordinary transactions; when a match is found, the link is made between the rewarding transaction of the miner and the spending transaction. Then, a well-known heuristic from [17] is applied, such that all inputs to a transaction belong to the same user. Two scenarios are identified in [3] for pool-hopping detection a simplistic one and a realistic one. Detection is qualified on a simplistic scenario, which considers rewards distributed at the end of the round in which they are earned; hence a reward is received in each round. A miner is placed in the rounds corresponding to the received reward, so the miners' work schedule is obtained by simply ordering the rewards. In the realistic scenario, on the other hand, even though a rewarding method is based on rounds, the reward is actually paid once a certain minimum threshold is reached. A miner's work can be placed in a time interval called 'epoch'. Only in the round that ends the epoch, one can be sure of the miner's presence since, thanks to it, it has reached the minimum threshold for the reward.

3. Pool-hopping Detection Framework

As anticipated, our pool-hopping detection framework is structured in four main modules, or Steps, we synthetically describe as follows:

Step 1 : Revenue Stream Distribution.
In this module, we model the revenue stream distribution process to link the rewarding transactions to a set

of Bitcoin addresses, such that then each address can be assigned to one or multiple mining pools.

Step 2 : Miners Identification.
In this step, we enhance and contextualize an existing heuristic at the state of the art to group addresses controlled by the same user, i.e. the so-called backbone miners.

Step 3 : Multiple Rounds Rewarding.
The goal of this step is to produce, for each miner and each pool it worked for, the mining schedule, so as to characterize its behavior.

Step 4 : Hoppers Detection.
Exploiting the outcome of Step 3, this steps determiners which miners are hoppers, and classify them as either intra-epoch hoppers or cross-epoch hoppers, saving the necessary information to further analyse their behavior.

Each step is described in the following four sections.

4. Step 1: Revenue Stream Distribution

In order to develop a refined detection of the pool-hopping phenomenon, it is necessary to investigate the dynamics governing pools and users within the system. Starting point of the analysis is the identification of the users who carry out the mining activities, namely the ones we referred to in the background section as *backbone miners*. For a ledger observer, an ordinary user and a miner are indistinguishable, since both are pseudonymous and send and receive transactions in the same way. Given an address, it is not possible to directly determine which user it belongs to, nor the role this user plays in the system. Since we want to understand whether a miner is working and moving from one pool to another, each address receiving a reward must identify a single miner. Therefore, before proceeding with multiple-input clustering heuristics, as done in [3], we need to collect all addresses that are certain to have participated in the validation process.

4.1. Identification of rewarding transactions

Unlike ordinary users, miners receive rewards for the work they performed. To identify miners, one must understand the dynamics that, within a pool, govern the distribution of revenue to each miner, after associating a reward with the miner. The mechanism involves the pool constituting and transferring what we have introduced as rewarding transactions. It is in the output of these that the addresses used by the miners are certain to be found, addresses with which they have contributed to the validation process and to which the pool sends the reward earned.

Although the purpose of rewarding transactions is different from ordinary transactions, they are constructed in exactly the same way and there are no structural features that distinguish them. The issue therefore concerns not only the identification of miners, but also the identification of reward transfers. Even though there are no specific attributes for their identification, just by looking at the ledger it is possible to notice particular transactions that may play a role in the distribution of rewards. Indeed, pools send rewards to numerous miners with a single transaction, which consequently results having a particularly high number of outgoing addresses. For example, Ant pool rewards its miners by creating large transactions that always have 101 output addresses. The amount that each address receives is in the order of mBTC as the reward is sent immediately after the payout threshold is reached, that is 0.001 BTC [3] [28]. Therefore, if a transaction has many output addresses, but it sends amounts above the range of a payout, the transaction is not a rewarding one.

Algorithm 1 is a pseudocode of this step; it receives as input the *ledger* L listing ordinary transactions, within which rewarding transactions are searched. Table 1 introduces the notations for this and following algorithms.

Algorithm 1 Matching

```
Input: ledger $L$
Output: matching structure $D$
$maxP \leftarrow$ MAXPAYOUT(L)
$minT \leftarrow$ MINTRANSFERS(L)
for $t \in L$ do
    $d[hash] \leftarrow t[hash]$
    $d[leaf] \leftarrow \emptyset$
    $d[od] \leftarrow$ OUTGOINGDIR(t)
    $d[$ISR$] \leftarrow$ ISREWARDING($t, maxP, minT$)
    for $id \in t[tIDs]$ do
        $d[pr] \leftarrow$ LINKTOPREVIOUS(t)
        $D[id] \leftarrow d$
    end for
end for
```

4.2. Characterization of pool revenue streams

Once a rewarding transaction is identified, the next step is to determine which pool is distributing the reward. No attribute of the transactions retains this information, so it is necessary to find a strategy to trace it back to the source pool. As studied in [18], most pools follow a distinct payment pattern to distribute rewards. In contrast, the analysis performed in [3] relies on the assumption that reward transactions are exactly successive to the coinbase ones. Currently, this assumption has to be disregarded, because although it might have been considered true in the past, it does not comply with what can be observed on the ledger any longer [18]. Indeed, the transactions that follow the coinbase ones are internal to the pool and not yet rewarding. It is likely that many of the addresses that were considered belonging to miners are actually addresses used internally within the pool to manage their own wallets.

Considering a coinbase transaction, i.e., the one sending newly generated coins to a pool that has completed the validation of a block, the outgoing address is the one the pool uses to receive payments from the system, which we now refer to as the 'pool reward address'. The pool uses this address to publicly receive mining revenues and, as it is pub-

Table 1
Algorithms functions and notations

Notation	Description of the function
isR	Boolean value, True if the transaction is a rewarding transaction.
od	ordered record of values, showing 0 if the outgoing transfer in that position is above the range of being a payout, otherwise 1 if it is in the range.
pr	the position that the income transfer holds within the previous one's outputs. This number keeps the elements in the lists ordered respectively to the order in which the transfers are store in the previous transaction.
$leaf$	holds reference to the shortest path a coin takes to fund it. This attribute is initially deactivated, and activated when the transaction is encountered for the first time during the visit of a tree.
S - Senders set	it matches each address that sends a transaction with all transactions sent from it. For each address, S gives the list of all hashes with that address as an input. S is filled out by reading all the transactions on the ledger, and when a CoinJoin transaction is encountered, it is ignored and not uploaded in the structure for any of its input addresses.
T - Transactions set	references all transactions $t \in T$; for each t, its hash is used as the key, the retrieved value is the entire transaction.
CS	Coinbase transactions set.
D	Matching structure that keeps the reference of all consecutive transactions from a given one, by matching its hash with the IDs inside them.
R - Rewarding set	contains all rewarding transactions, accessible by hash number.
M - Miners set	contains all miners identified in the system, as unique collections of addresses.
U - round set	holding the schedule of all rounds completed by each pool sorted chronologically.
E - epochs set	references all hoppers' epochs, indexed by miner and pool.
I - intra-epoch hoppers	indexes all intra-epoch hoppers.
C - Cross-epoch hoppers	indexes all cross-epoch hoppers, each referring to its working epochs. The reported work epochs are only those that are found to be simultaneous with others of the same user in different pools, shown in a 2-by-2 correspondence.
W	windows set.
maxPayout(L)	calculates the upper bound of the of the distribution composed by all payouts in particularly big transactions. The outcome is the value above which the amount of a transfer is not considered a miner payout.
minTransfers(L)	calculates the median of the number of transfers in particularly big transactions. The outcome is the minimum number of transfers, whose amount are payouts to miners, to consider a distribution of rewards.
outgoingDir(t)	checks which amounts are above the range of payouts and which not. The ordered record is filled respectively and assigned to d.
isRewarding(t)	counts the number of outgoing transfers and how many of them are payout, then it assigns isR consequently.
linkToPrevious(t)	saves the position associated with the ID corresponding to the key by which the element will be referenced.
tobeFwed($d[od]$, $n[pr]$)	checks if in the ordered record if the amount in position pr in the previous transactions is a payout or not.
calcOwnership(r)	checks which pool's coins reached the transaction in the shortest path & assign its ownership to a pool accordingly.
collectHashes(S[**a**])	adds the current reference hashes in H, with those yet to be visited, as all sent by the same user, but only if they have not already been associated with the miner.
getSenders(T[**h**])	retrieves all the input addresses of the transaction corresponding to hash h.
getPools(m, R)	returns all the pools miner m has received rewards from.
setIntervals(m, p, R)	it collects all the rewarding transactions of miner m in pool p, ordered chronologically & returns the reward intervals.
Q3(I)	calculates the third quantile of the reward intervals distribution.
getStart(t)	given a transaction t, it sets the beginning of the epoch with the timestamp of the previous transaction.
adjCoolDown($t+1$)	given a transaction t, the next rewarding transaction $t+1$ is the one closing the epoch of work between t and $t+1$. The function adjust the conclusion of the epoch based on the time of rewarding transaction $t+1$, but subtracting the cool down interval. Approximation of the interval to be subtracted is assumed constant and equal to the average validation time of 100 blocks.
createEpoch($start$, end)	creates the epoch interval with respect with the calculated start and ending moments.
singlePool(m, E)	returns True if the miner present epoch in a single pool; performed by parsing all epoch referenced by the miner.
consEpochs(m, E)	returns all the epochs carried out by m and stored in E, sorting them chronologically according to their completion.
getEpoch(m, n, $\bar{E}$)	returns the epoch at index n, between the ones referenced by miner m.
differentPool($e1$, $e2$)	returns True if the epochs currently compared, $e1$ and $e2$, are placed in different pools.
getOrdrdRWTs(i, R)	collects are rewards received by intra-epoch hopper i in R and sorts them chronologically.
samePool(t, $t+1$)	returns True if one reward and the following, received by the hopper, are sent from the same pool.
increment(i, $t[pool]$)	increments the number of times intra-epoch hopper i has left the pool of the reward being analysed. The updates is done over I.
getRounds($start$, end, $e[pool]$, U)	returns all the round, sorted chronologically, that a given pool concluded in the time interval $start$ - end; the rounds are retrieved from the global round schedule U.

licly known, it is used on a long-term basis. After the transfer of bitcoins to a pool via a coinbase transaction, the distinct payment pattern is characterized by how the pool handles the exchange of coins in its wallets, from the receipt of newly generated coins to the payment of rewards to miners. We will refer to all transactions that occur within a pool from the first phenomenon to the second as 'revenue streams', as qualified in [18]. We refer to 'collector addresses ' for the addresses that are used by the pool to manage, maintain and move large amounts of bitcoins via revenue streams and send payments to pool members and external parties when necessary. When a transaction sends funds from inside the pool to the outside, one of the output addresses is the 'changing address ' which allows the pool to retrieve the bitcoins not wanted to be sent. The exit point of a stream are the reward transfers, identifiable as being within the payout range. These addresses belong to the miners and from them the streams outside the pool originate, i.e., those that describe how a miner spends its rewards. In order to keep the analysis within the stream of a pool, it is necessary to consider only the transactions that arise from collection addresses or changing addresses, which transfers remain above the payout range.

Thanks to the reward distribution analysis described so far, when a rewarding transaction is in the stream, we know that some of the bitcoins transferred by the transaction were generated for the pool receiving the coinbase at the beginning of the stream. Therefore, given a coinbase transaction, the pool for which the new coins were generated is known, then by following the transactions we maintain information about which pool sent them. However, this is still not enough to say that the pool sent the transaction or which other pool did. Indeed, the pool sending a transaction is unique, but the coins financing a transaction it may have originally been generated for different pools and then shuffled through the system up to this transaction.

Algorithm 2, Revenue Stream Visiting, details this step. It requires to start from a coinbase transaction and visit all subsequent ones. Given the structure of a transaction, however, it is possible to link it to previous ones in the backward direction and not in the forward direction. For this purpose, it became necessary to build a $Matching\ Structure$ (D) that keeps the reference to all consecutive transactions from a given one, by matching its hash with the IDs inside them. So, given a transaction we need to collect all following transactions, funded by it. Each ID stored within a transactions uniquely identifies the transaction sending funds to it. Even though they identify uniquely the previous transactions, IDs are repeated in different transactions, namely in all the outputs of the previous one. Each ID correspond to a unique transaction hash, thus they can be used as keys in the structure. Scanning the transactions, each key collects all the transactions that holds that ID. Eventually, accessing a key we reference a unique transaction and as value we retrieve the list of all transactions one following it. To save memory, the data of a transaction maintained in the structure is minimized, keeping only the information that cannot be replaced by attributes indicating which action to perform.

The attributes are assigned while reading the ledger and filling the matching structure. isR is set to True when the number of outgoing transfers is above a certain value, but it only counts a transfer if the amount is in the range of being a payout. Both the minimum number of outputs and the payout threshold are set in a preliminary step, where, considering only large transactions, the average number of outgoing transfers and the average transfer amount are calculated. The resulting values for considering a transaction to be a rewarding are 97 for the minimum number of transfers and 0.045 BTC for the maximum for an amount to be a payout. When the matching procedure is completed, algorithm 1 return in output $Matching\ Structure\ (D)$.

Algorithm 2, visits each tree originated by a coinbase. It takes in input the *coinbase set* CS, that is the data set listing all coinbase transactions, roots of the transactions trees. Since the detection is carried out in the major pools, the coinbase transactions are classified as being sent to one of the pools of interest and as being sent to minority pools. At this stage we must also consider the second category because, in order to avoid possible misallocations on the ownership of the transactions, it is necessary to analyze the involvement of the streams of the secondary pools in correspondence with those of the pools of interest. Algorithm 2 starts accessing the hash of the coinbase transaction as first key generating the tree. The tree visit only goes from one level to another when all transactions in that level are visited. (V) holds all the transactions to be visited in a level. While visiting a level, all branches of the next level are added in a support structure (N), waiting for this level to finish being visited. Algorithm 2 manage this procedure with the following instructions:

- Increment rule: when V is emptied from the transactions of the current level and new branches have been found and added in N, the level is incremented and new transactions to be visited are added to V .
- Stopping rule: it is the rule that ends the visit of a tree when no more transactions are added to V, meaning that in the current level no branches has been found to visit.

When a transaction is visited, the Follow function is called, algorithm 3. It receives as input the transaction hash, the pool associated with the tree, and the level at which the transaction is encountered. To visit this transaction the hash is used as the access key in the matching structure. Not all subsequent transactions (n) have to be considered as new branches for the next level.

Conditions 1 and 2 in Table 2 have to be True in order to return a branch with the ones to visit. If condition 2 is False, there is no need to continue on this branch, since all following transactions would also be visited at a deeper level than already done. Instead, if the condition is True, algorithm 3 updates the leaf of this transaction with the new, now shallower, level than the one already stored, if any. At this point, if the transaction is a rewarding one, the pool and depth level in the tree are associated with this transaction. All n transactions in the key, which satisfy the conditions, are returned to algorithm 2 as new branches to be visited in the next level. When all transactions are visited, each rewarding transaction is associated with the pools to which the different funds in the transaction were originally sent. The pool that reaches the transaction with the shortest path from the generation of some of its coins is the one most likely to have sent the transaction, since it owns the least spent coins in the transaction, so it is assigned as the authoring pool for the transaction. In case of a tie, so if two pools reached the transaction at the same depth, which is also the minimum level of all trees reaching it, which is unlikely, the pool ownership is assigned to the one that owns the majority of the least spent coins in that transaction. Algorithm 2 returns in output the data set listing the identified rewarding transactions associated with a pool, i.e., the rewarding set $\boldsymbol{R}$.

Appendix A gives an example of how the algorithms perform this step of the strategy.

Algorithm 2 Revenue Stream Visiting

```
Input: coinbase set $C$, matching structure $D$
Output: rewarding set $R$
Initialize $V \leftarrow \emptyset$
Initialize $level \leftarrow 0$
for $c \in CS$ do
    $V \leftarrow c$
    $level \leftarrow 0$
    while $V \neq \emptyset$ do                          ▷ Stopping rule
        $N \leftarrow$ FOLLOW($V_1$, $D$, $c[pool]$, $level$)
        delete $V_1$
        if $V == \emptyset$ then                        ▷ Increment rule
            $level \leftarrow level + 1$
            $V \leftarrow N$
        end if
    end while
    for $r \in R$ do
        CALCOWNERSHIP(r)
    end for
end for
```

Algorithm 3 Follow

```
Input: $d$, matching structure D, $pool$, $level$
Output: next level Txs $N$
Initialize $\mathbf{h} \leftarrow d[hash]$
for $n \in D[\mathbf{h}]$ do
    if TOBEFWED($d[od]$, $n[pr]$) then                  ▷ Condition 1
        if $n[leaf] \geq level$ then                   ▷ Condition 2
            $n[leaf] \leftarrow level$
            if $n$[ISR] then
                UPDATEREWARINGSET($n[hash]$, $level$, $pool$)
            end if
            $N \leftarrow n$
        end if
    end if
end for
```

4.3. Pool ownership assignment

Following revenue streams, one could assume that the bitcoins in a transaction are owned by the pool receiving the coinbase at the beginning of the stream. As a matter of fact, after numerous transfers, one cannot be sure that coins still belong to the pool they were generated for, as it may have happened that they were used for payments outside the pool. Indeed, a coin is moved within the same pool until it is needed for a payment, then it is sent out of the pool to a miner or other entity. In turn, a second pool can use the coin received from the first to pay its rewards. Although revenue stream analysis is aimed at studying Bitcoin transfers between wallets belonging to the same pool, avoiding tracking transfers of wallets outside of it, it may happen that a transfer above the payout range is made outward and is thus momentarily interpreted as an internal transfer. This eventuality translates into the fact that by following the revenue streams of coins generated for two different pools, it may happen that the same transaction is part of both streams. In this case, it is contradictory to assign the transaction to one pool rather than another.

To get to the point of defining which pool really sent a rewarding transaction, we need to analyze all the streams through it. By tracking all the coins that start from the coinbase transaction and pass through the transaction in question, it is possible to figure out which coins in that transaction come from a longer path and which come from a shorter one. Using the same procedure, it is also possible to determine which portion of the funds funding the transaction were originally generated for one pool and which for another. Thus, given a transaction and several assignee pools of the bitcoins transferred with it, it can be argued that the transaction was actually handled by the pool with the least reused coins among those present. This assertion is acceptable because it is unrealistic that a coin, immediately after being generated for one pool, is sent directly to another pool. This part of the analysis can be defined as a procedure to understand when the ownership of certain funds, originally assigned to a pool, can be considered valid and when it should be considered overwritten. Because of the ownership assignment procedure, the link between a pool and a profitable transaction is now known. This establishes a key relationship between miners and pools. In other words, given a rewarding transaction, we know that the output addresses belong to the miners, so, knowing also the pool that sends the transaction, allows us to understand in which pool the miners are working.

4.4. Transactions trees and forests

Going deeper in the procedure of analyzing the revenue stream of a pool, the stream can be considered as a tree whose root is the coinbase transaction. Following a transaction in the tree refers to the task of selecting which of the outgoing transfers are payouts and which are not, starting a new branch for each transfer that is not a payout. All the trees generated by coinbase transactions constitute a forest. A transaction can be the overlapping point of different trees, even

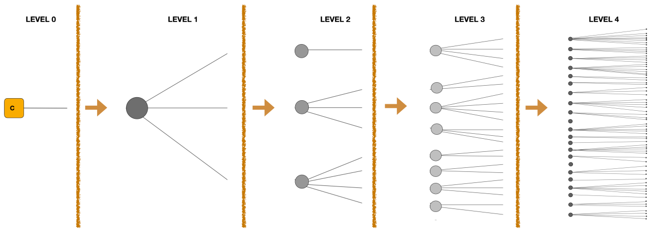


Figure 3: Figure shows the distinction between levels in a tree.

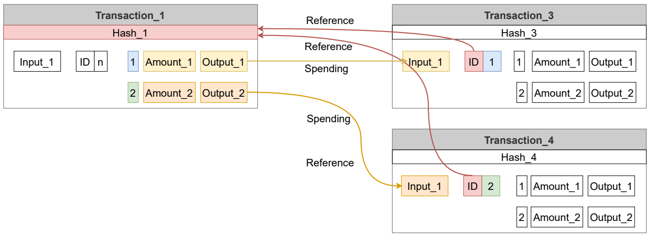


Figure 4: Chain of transaction with focus on the reference to the previous transaction.

if owned by different pools. The leaves of the trees are the payout transfers from the rewarding transactions, i.e., those that are not chosen to be new branches. Since we must keep track of the number of transfers that each newly generated coin undergoes, it is necessary to use a strategy that allows us to count as fast as possible at which level of depth the coins arrive in the tree. Hence, our logic visits the tree by 'levels ', starting from the root, towards the leaves. Only when all the transactions of a level are visited, the successive level, composed of the branches that depart from the transactions of the current level, begins to be visited. Figure 3 aims to show that to visit completely a level at a time is much faster than to follow all the consecutive transactions that depart from a fork and then to return back to that fork and to visit the other branches.

To explain the way the tree is filled, we need to recall the structure of a transaction, explained in the section 2. In a transaction, there are two key attributes that allow us to bind one transaction to the previous one. As shown in Figure 4, for each incoming address, the transaction stores an ID that is the hash value of the previous transaction. Given a transaction, the link to the previous one in the tree is created by searching for all transactions with hashes corresponding to the IDs stored within the transaction, thus obtaining all previous ones to the current one. The second key parameter is the position number paired with each ID in the transaction. This number corresponds to the position where the input address is between the outputs of the previous transaction that has the paired ID as a hash.

5. Step 2: Miners Identification

This section presents the miners identification task for the pool-hopping detection algorithmic framework. Its goal is to group addresses controlled by the same user. Starting from the addresses identified by the revenue stream analysis, we can identify the miners working in the pools of interest as large entities composed of homonymous addresses. The procedure was already presented in [3], with the application of some simple but effective heuristics we elaborate next, in compliance with the procedures applied so far.

5.1. Address grouping heuristic

The heuristic yields a clustering of Bitcoin nodes [17]: if two (or more) addresses are inputs to the same transaction, then they are set as controlled by the same user. The effects are transitive and extend well beyond the inputs of a single transaction; for example, if we observe one transaction with addresses A and B as inputs, and another with addresses B and C, then we conclude that A, B, and C all belong to the same user. In order to identify the addresses that a single miner uses in the system, transitivity is fundamental indeed.

Let us consider a rewarded address obtained as described in the previous section. First we look for the transactions that have this address in common among the senders. The miner owning the initial address is the creator of the obtained transactions, and all the sender addresses of these transactions belong to him. For each transaction sent by the miner, the input addresses may be already known as homonyms of the miner, or they may be addresses we did not know as belonging to this miner yet. In the second case, the search continues for all transactions that have those addresses as input. Thus, the process iteratively populates the set of transactions that the miner sends and that are related to it by the transitive property. All addresses that uniquely identify the miner are collected accordingly, i.e., all senders of transactions for which the transitive property is verified.

Attention should be paid to the difference between the group of transactions just described and a stream of earnings, subject of the previous section. Here transactions are not searched consecutively at each funding, but are united by homonymous addresses and ordered respectively upon encountering a given address and associating it with a user. For example, assume addresses A and B are the senders of a transaction and addresses B and C are the senders of a second transaction; even if we consider first the first one and then, thanks to the common address B, the second one, one cannot assume that the second one is temporally subsequent to the first transaction.

The strategy just described allows us to be sure that we have found all addresses linked together by the transitive property and related to the same user. This ensures that there are no cases in which the same user is identifiable by two distinct groups of addresses; in other words, in which there is even one common address shared by two groups. The only case that is overlooked is when a user is considered to be two distinct entities, presenting no common address. This is what would happen if two groups of transactions sent by the same user never reuse an address of the other group. In this case, the heuristic cannot connect the two groups of transactions and two identities are assumed. Indeed, a miner could use always different addresses independently and there is no certainty that one sends transactions always with at least an addresses already used. Therefore, the heuristic cannot always collect all addresses governed by a user, but at least all the ones linked by the transitivity property. Therefore, the deriving measurement of miners population and hoppers is to be considered as an upper bound.

Moreover, the heuristic does not consider coinJoin transactions, i.e., transactions for which two or more users can organize to send a single transaction by sharing each other's funds. These addresses appear to be senders in the same transaction, so they are joined by the heuristic in the same group. Consequently, these transactions must be excluded before applying the heuristic, otherwise what are actually two distinct users would end up being considered as one single user.

An alternative heuristic exists [12], but we cannot use it. It joins the inputs of a transaction with the *changing address*, all identifying the same miner: the changing address is controlled by the same user as the input addresses; i.e., for a given transaction t, the controller of inputs(t) also controls the one-time change address in outputs(t), if such an address exists. This approach therefore could compensate for the missing grouping above mentioned, making transitivity to work also in the case the addresses associated to an entity are not reused. However, we cannot adopt this logic because our method is meant to be applied to work not on the full ledger, but a fraction of the ledger for scalability reasons: hence it is not possible to identify which of the addresses in a transaction is with certainty a changing address, hence owned by the sender.

It is worth remarking that we cannot take into account in the mining identification any attack (e.g., man-in-the-middle) happening on the application programming interface between the miner and the mining-pool, aiming at stealing the mining rewards. Such attacks would, in any case, result as miners with a single address, hence not hoppers, in the public blockchain.

5.2. Detailed algorithm

We detail the algorithm we used to implement the heuristic to identify and group the addresses controlled by the same user. It creates the registry of miners, that is composed of sets of addresses (not repeated ones), each set representing a single and unique user. In a preliminary stage of the procedure described hereafter, the S, T and R data structures described in Table 1 are created to support the execution and make it as simple and quick ad possible.

Algorithm 4 details the heuristic; it receives S, T and R as inputs and starts iterating through all addresses in R, recipients of a reward, each identifying a miner (m). The construction of S is useful because, given an address from R, the procedure retrieves directly the hashes of all transactions that show it within their inputs. These transactions are not only sent by the same user, but specifically from the same address (a_1). To keep track of all transactions sent by m during its identification, their hashes are held be support structure (H). Each iteration starts with an address from R and initializes H accordingly.

In the heuristic, all sender addresses of transactions in H are homonyms, but they are also homonyms with those of the other transactions, since they all share a_1. Then the algorithm gets from T the homonymous senders of each transaction corresponding to H. These new addresses (a_2) owned by m are accessed on S. The procedure looks in S to see if they have sent any transactions not among those initially identified as being sent by m. If so, the transitive property is verified again for these transactions, which are related from the previous ones by at least one common address a_2. The corresponding hashes are added to H and the inputs are collected in its group of homonyms. Such hashes are kept in H for the period of time they are already known to have been sent by m, but whose inputs addresses have not yet been accessed. If one of the addresses collected during the identification of a miner is further present in R, then the collection of homonymous addresses will not be performed for such address. Indeed, there is no need to access it again in S, since all transactions referenced in S by that address; the address itself, can only belong to a single miner and have already been recognized and associated with one. Also, once a transaction is visited, all of its inputs are considered, so accessing it again would result in addresses already known for m. All addresses visited and assigned to a user are maintained in the support structure A and are no longer accessible.

Condition 3 and 4 (Table 2) govern the algorithm in the access to S.

Since a transaction may have several input addresses, a hash may be encountered multiple times during the identification of m's addresses. Therefore, if a hash was already visited and it is known to have been sent by m, it is not added to H. The *Stopping rule* simply terminates the collection procedure for m when all transactions linked together by the transitive property have been visited, thus when homonymous addresses of m have been collected.

Appendix B gives a numerical example of the algorithm execution.

6. Step 3: multiple rounds rewarding

As described in Step 1, we can tie each reward present on the ledger to the pool that sent it. In Step 2, we collect all the reward-receiving addresses into user-groups, each uniquely identifying a miner; in this way we can (i) know from which pool the rewards of a given miner arrive and at what time, and (ii) place it with certainty in each pool from which it has received a reward (as it must have worked there for at least a certain period to receive it). Nonetheless, being placed in several pools is not enough to study the simultaneous working of miners, as the work periods are different in each pool and pools overlaps are not directly noticeable. We need to access the precise starting and ending moment of a miner contribution to determine whether it has dynamically moved between pools during ongoing work periods. The goal of Step 3, therefore, is to present a strategy to characterize and quantify the work done by a miner in a pool in order to allow the identification of concurrent working for different pools.

Table 2
Description of conditions used in the algorithms.

Condition 1	it checks if the outgoing transfer is less than or equal to the payout range, since we want to follow only those transactions that are still handled within the pool. For this check the position of the outgoing transfer in the transaction currently visited is given by the *od* attribute of n
Condition 2	Visit only if it is a shorter path. Algorithm 3 checks that the transaction is met on this tree at a shallower level than done in the other trees.
Condition 3	it checks if the current address from R is not already associated with a miner and thus present in A. In addition, for each address encountered as an input to a transaction, it checks if it has already been accessed in S, thus associated with a miner and present in A.
Condition 4	it checks if an address receiving a reward is actually sending transactions. If not, that address has no homonyms and the user is identifiable only by that single address.
Condition 5	it ensures that the length of the period resulting from the adaptation is no longer than the limit duration computed at the beginning. Otherwise, the start is set relative to the maximum length of an epoch considered realistic
Condition 6	it checks if the miner presents epochs always in a single pool
Condition 7	it checks if the epochs were carried out in different pools. If False, they identify two phases of ordinary work in the same pool.
Condition 8	it checks if the epochs, placed in different pools, are simultaneous. To be such, the beginning of the current must be earlier than the end of the oldest, meaning that the oldest was not over before the second began.
Condition 9	it checks if the miner did not show any overlapping epochs; if so, it is deleted from C and added in I. Indeed, it is an intra-epoch hopper, showing epochs in different pools, but not simultaneous.
Conditions 10, 12	it checks which epoch starts before the first finishes.
Conditions 11, 13	it checks which epoch finishes first.

Algorithm 4 Miners identification

```
Input: rewarding set $R$, senders set $S$, transactions set $T$
Output: miners set $M$
Initialize $A \leftarrow \emptyset$
for r $\in R$ do
    if r $\notin A$ then                              ▷ Condition 3
        if r $\in S$ then                             ▷ Condition 4
            $m \leftarrow$ r
            $H \leftarrow$ collectHashes($S$[r])
            while $H \neq \emptyset$ do               ▷ Stopping rule
                $h \leftarrow H_1$
                $s \leftarrow$ getSenders($T$[h])
                for a $\in s$ do
                    if a $\notin A$ then              ▷ Condition 3
                        $H \leftarrow$ collectHashes($S$[a])
                        $A \leftarrow$ a
                    end if
                end for
                delete $H_1$
            end while
            $m \leftarrow A$
        else
            $m \leftarrow A$
            $A \leftarrow$ a
        end if
        $M \leftarrow m$
    end if
end for
```

6.1. Mining dynamics

In [3], pool-hopping is detected by considering a simplistic case, in which pools reward miners with a per-round logic. We recall that a round is the period of time that occurs between two subsequent validations performed by the same pool. After this period, the rewards are distributed to the miners that worked during it. The per-round rewarding logic means that it is possible to place a miner in each round for which it worked, due to the reward resulting from it. The simplification lies in the fact that the reward methods used at that time, as well as those used nowadays, do not really send rewards at the conclusion of each round or immediately after a share submission. Indeed, the system requires pools to establish a minimum reward threshold, which for many pools is 0.001 BTC. In reality, the reward for a single share can be much lower than this figure [19]. For example, let us consider a block reward (B) of 6.25 BTC, share difficulty (D) equal to 1, network difficulty (D) 240000 and 2% fee (f). It results that each share posted would yield on average 26 BTC. Therefore, we can assert that, in a realistic scenario, a miner has to send many shares before reaching the threshold and receiving a reward. Considering PPLNS, for example, the rewards are not sent upon the fulfillment of N, and there is no relation between N and the threshold, so the miner can still be far from reaching it even if N shares have already been submitted. If N is large enough, the proportional coefficient is toward the value of the miner's hash rate and the pool mostly wins in the validation of blocks the miner worked for, then the threshold is likely exceeded with each reward. Conversely, if N is reached, but the reward is below the minimum threshold, more shares are required, so the work period to get the reward will continue in the following rounds.

The time a miner has to work until it reaches the threshold and receives a reward is called 'epoch'. Although some pools use both immediate rewarding methods, such as PPS, and non-immediate ones, such as PPLNS, there is no need to distinguish which method was used to establish a reward, since each of these methods are eventually governed by the per-epoch payment system. At the same threshold, epochs rewarded with PPS will last less than those rewarded with PPLSN, since in the first case every share presented is rewarded, while in the second only rewards for blocks in the chain increase the threshold. For a deeper understanding,

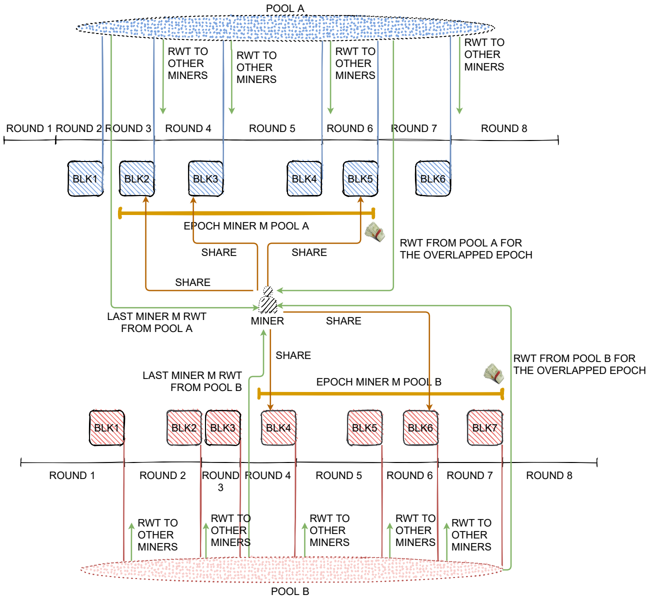


Figure 5: Representation of the rewarding dynamic for a single miner, two epochs in different pools.

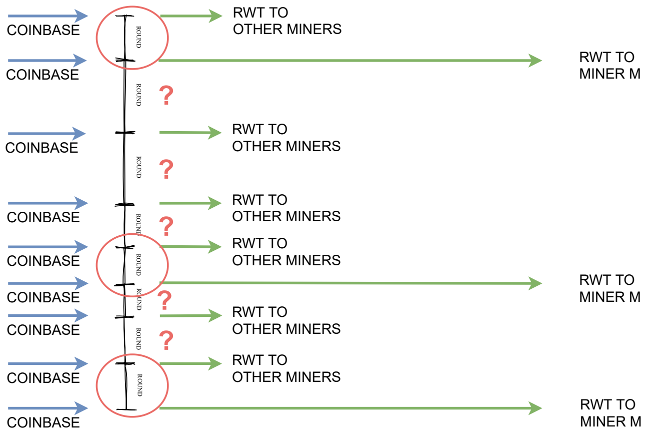


Figure 6: Hopping dynamic in a minimum rewarding threshold environment.

one can compare each rewarding method in both the simplistic and realistic scenarios. For the pool which implement PPS, PPS+ and FPPS - hence instant rewarding - if the system would not impose a minimum reward threshold, looking at the rewards received by miners would allow to place them exactly in the rounds they participated in. This would fall into the simple case of [3] and per-round rewarding algorithm would work just fine. For PPLNS, unlike the immediate reward case, even neglecting the existence of a minimum reward threshold, this would not fall into the simple case, since N is not bounded by rounds. Given the existence of a minimum reward threshold, as said in the previous work, miners working in pools implementing these rewarding methods need to be studied following a per-epoch analysis. Understanding the epochs of each miner is key for identifying hoppers for all types of rewarding method. If a miner's epochs in the different pools overlap, it means that it was present in all at the same time, jumping from one to another dynamically and at will. Figure 5 shows all the elements involved in the dynamic. Given a miner, one can see the shares it sends to contribute in the validation process, the rewards it gets and the identification of its work-epochs in each pool.

6.2. Dealing with epochs

An epoch is an interval of time unequivocally related to a miner and indicates a cycle of work it performed in a pool. It is the sum of all rounds executed by the pool from the moment the miner started working for a new threshold until it was reached. The Bitcoin blockchain does not keep information about epochs, so, as we can see from Figure 6, the miner might have participated in all rounds in its epoch or just in some. Only in the last round its presence is certain, since thanks to it, it reached the minimum reward threshold, but beyond that nothing can be determined.

To place miners in their working rounds, we would need to know which user is sending each share and, further, which shares helped validating which blocks. The only available information comes from the rewards each miner receives at the conclusion of the epochs. Since it is necessary to construct an analysis based on epochs as close to reality as possible, we need to identify what a realistic epoch duration might be and, from there, describe the epochs in terms of their beginning and ending moments. Consequently, the problem becomes finding a way to assert, with enough confidence, what is the initial round of the miner's contribution and the final one.

Based on the definition of an epoch, we know that its conclusion coincides with the last round within it. This round is the one in which the miner reaches the payout threshold, so we can initially assign the time of the rewarding transaction to the instant of conclusion of the epoch. As reported in [25], there is a delay interval from the block reward, given to a pool via coinbase transaction for the conclusion of a round, to when it is available to be spent. This interval is called 'cool down' and corresponds to the time taken by the system to validate 100 blocks. This implies that, given an epoch, if the rewarding transaction uses funds provided by coinbase less distant than the interval of cool down, then with certainty the rewarding transaction is delayed. At worst, if funds are used from the coinbase that rewards for the last round of the epoch, the delay of the rewarding transaction is equal to the time it takes for the full cool down interval. In contrast, if the rewarding transaction uses funds previously received from the pool, it is sent immediately at the conclusion of the round. Since we do not know whether the funds used for a rewarding transaction are those that come from rounds completed within the epoch that the transaction is rewarding or from earlier rounds, it is not possible to tell whether the rewarding transaction is actually delayed and especially by how much compared to the conclusion of the last round. Nevertheless, even just using a small portion of the funds received from a recent coinbase transaction are enough to delay the rewarding transaction by part, if not all, of the

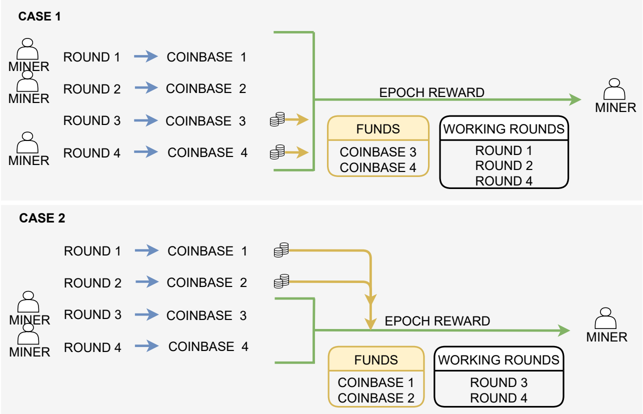


Figure 7: The figure shows the funds that support a rewarding transaction.

cool down interval. Moreover, it is quite likely that the pool will use at least some of the funds obtained from the rounds for which it is repaying the miner. Therefore, we consider the transaction to occur 100 rounds after the round closing that epoch. The average confirmation time of a round is 10 minutes, so we can take the time to subtract from the rewarding transaction as constant, thus obtaining the instant when the epoch ended. Instead, about the beginning of epochs, several assumptions can be integrated by our method.

The first assumption is from [3]. It considers the start of an epoch with the first coinbase transaction providing funds for the transaction that pays that epoch. Particular attention must be paid to understanding the distinction between two very different roles that a coinbase transaction can play with respect to a reward:

- One role is played by the coinbase transactions that provided the pool with the funds it needs at the time of sending a rewarding transaction, i.e., those that are inputs to the rewarding transaction and linked to it by the hash.
- The second role is taken on by coinbase transactions that are paid to the pool at the end of each round for mining blocks, for which the collaborating miners will be remunerated later on.

Indeed, the pool can spend the bitcoins received from a coinbase for validating a block in advance compared to paying the miners who worked in its validation. In this case, represented in Figure 7 with case 1, the miner is payed with two recent coinbase funds, even if it worked also in previous rounds. Conversely, the pool could also use old coinbase funds already in its wallet to pay for a current validation (case 2).

The distinction presented above makes the first assumption unusable, since a rewarding transaction pays several miners, so this would force all of them to have the start of their epochs with the same round. Also, considering the first transaction funding the reward as the starting point of an epoch is incorrect, since it provided the funds currently used by the pool, which are not tied to the work done by the miners rewarded with this transaction. Ideally, the correct coinbase to consider as the starting point of an epoch should be the one that paid the pool for the round at the beginning of the miner's epoch. Unfortunately, no correlation with this coinbase transaction is maintained by the blockchain, so this assumption must be discarded.

It is worth mentioning another possible assumption, which however we have to discard. One could set the beginning of an epoch at the previous rewarding transaction received by the miner, as it marks the last threshold it reached. As this would be a rough approximation, we discard it; indeed, a miner could have taken a break after having reached the last threshold and, in this way, the duration of a possible pause would be considered part of the current epoch.

Finally, the strategy to solve the approximation is understanding whether the time elapsed from the last rewarding transaction is reasonable and, if not, assigning to the epoch an average duration value. To calculate such a value, it is possible to leverage on statistical features, as several reward intervals have similar lengths and can be seen as a distribution. Given the distribution of intervals, the median is equal to the average of the intervals placed in the center of the distribution. Moreover, the maximum length of the epochs in the center of the distribution is known and it is the third quantile of the distribution.

It is used as the epoch upper bound, after which a epoch length can not be considered reasonable. Epochs with length beyond the limit take the upper bound itself as duration, since there is a high probability that a break was taken after the last reward.

6.3. Detailed algorithm

Algorithm 5, EPOCHS ESTABLISHMENT, focuses on the working schedule of each identified miner. In particular, it calculates the upper-bound to consider an epoch realistic and establishes the schedule accordingly. In input it receives R and M. In output, it produces the epochs set E, which indexes for each miner its work schedule in the pools in which it is present.

The first task of the algorithm 5 is to collect in I all miners' reward intervals. They result from sorting the reward transactions in chronological order and by pool. The time of a reward transaction is assigned as the conclusion of a reward interval and the time of the previous transaction as the beginning. The cool down delay is simplified because both transactions are delayed equally, as we consider it constant. The reward intervals in I are grouped firstly by miner and secondly by pool. Next, the third quantile of the distribution composed of all intervals in I is computed, excluding all above values. The result is the limit l, a value used to consider the length of a reward interval consistent with constant work activity, or to adjust to represent the epoch realistically.

At this point, the procedure iterates over I, accessing one miner at a time. For each miner iterates over the pool in which it has reward intervals. Then, for each miner's pool, the procedure iterates over the reward intervals to adjust each one to the realistic working interval, transforming it into an epoch. The *start* of the work period is assigned with the time of the first transaction of the reward interval and the *end* with the second with the newest transaction. A reward transaction is sent along with the end of the miner's work, but with a delay equal to the cool down interval. Therefore, the algorithm subtracts this interval from the time stamp of the transaction. Then condition 5 applies (Table 2). The only special case concerns the work interval terminated by the first reward received by a miner. There is no previous reward to recommend the length of the work done, so the epoch is set by assigning the maximum length of a realistic epoch. At this point, the beginning and end of each epoch are established and the final schedule of epochs, for each hopper in each pool, is written.

Appendix C gives an numerical example of the execution of the algorithm.

Algorithm 5 Epochs establishment

```
Input: rewarding set $R$, miners set $M$
Output: epochs set $E$
Initialize $I \leftarrow \emptyset$
Initialize $l \leftarrow 0$
for $m \in M$ do
    $P \leftarrow \textsc{getPools}(m, R)$
    for $p \in P$ do
        $T \leftarrow \textsc{setIntervals}(m, p, R)$
        $I \leftarrow [m, p, T]$
    end for
end for
$l \leftarrow \text{Q3}(I)$
for $i \in I$ do
    for $p \in i$ do
        for $t \in p$ do
            $start \leftarrow \textsc{getStart}(t)$
            $end \leftarrow \textsc{adjCoolDown}(t + 1)$
            if $end - start \gtrsim l$ then                ▷ Condition 5
                $start \leftarrow end - l$
            end if
            $e \leftarrow \textsc{createEpoch}(start, end)$
            $E \leftarrow [m, p, e]$
        end for
    end for
end for
```

7. Step 4: Hoppers Detection

For Step 3, we described the collaboration among miners in the pools through the characterization of work epochs, at the end of which miners get a reward. For an observer of the reward distribution, given the set of miners in the system, it is straightforward to identify which miners receive rewards from different pools. Specifically, each rewarding transaction that is consecutive to one coming from a different pool identifies a pool change made by the miner. This phenomenon consists of a jump from the pool of the oldest reward, to the pool of the second reward, which occurs at some instant between the two rewards. Some of the miners with rewards from different pools may have switched the pool of contribution without any strategic attempt. Alternatively, they may have participated in multiple pools by jumping from one to the other. Thus, there is a fundamental distinction to be made between two jumping behaviors.

- In one case, a miner performs occasional jumps, which can be seen as simple pool changes after completing a period of work. The epochs results in a non-overlapping pattern, meaning that it worked in different pools, but in uninterrupted periods, and never in both at the same time.
- In the second case, a miner actively jumps between two pools during concurrent rounds. This jumping behavior occurs when, looking at the epochs of the same miner, there are some that are placed different pools, overlapping each other.

The establishment of working epochs aims to resolve this distinction. Indeed, when analyzing the epochs of a miner, the key factor for which two epochs overlap is that the end of one epoch is later than the beginning of a second. Calculating the length of epochs as realistically as possible gains so much importance, as it allows us to identify overlaps more accurately.

7.1. Intra-epoch hoppers

Although the behavior of miners who do not exhibit simultaneous working between pools seems to impact less on pool hash rates, it can occur that they frequently switch the pool. We know that 'intra-epoch hoppers' do not appear to have performed simultaneous works in more than one pool, however they may have followed a strategic behaviour whereby, as the simultaneous working miners, they also hopped to exploit system changes to their advantage. If they are following such a strategy, their work period is at least spaced apart by a minimum interval so that no concurrent work is performed. However, if they change pool many times, they should be considered hoppers, since they are shifting their computing power continuously between pools. A pool change is identified simply by two rewards sent from different pools, and received by the same miner consecutively, meaning that the miner has jumped from the pool of the first reward to the pool of the second reward. The purpose of this step of the analysis is to access the frequency of jumps of intra-epoch hoppers in order to determine whether there is, or is not, an opportunistic achievement of rewards in their mining activity.

7.2. Cross-epoch hoppers

We define 'cross-epoch hoppers' the miners that have overlapping epochs in different pools. We can observe this when a miner finishes an epoch in a pool and it is already present in a second one. The overlapping of its epochs is the consequence of the hopper starting to work for the second epoch while it was still contributing to get the first reward. Also, to get the reward in the first, while it was also working in the second, it must have gone back and forth between

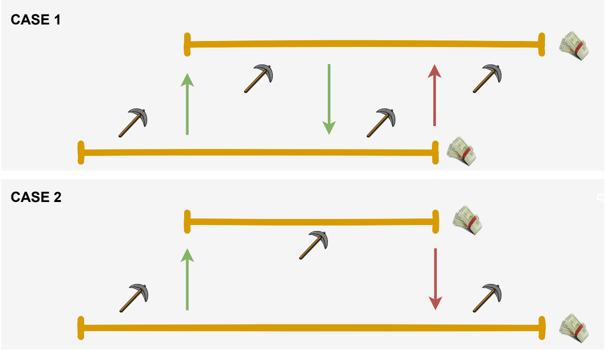


Figure 8: The representation shows the known jumps.

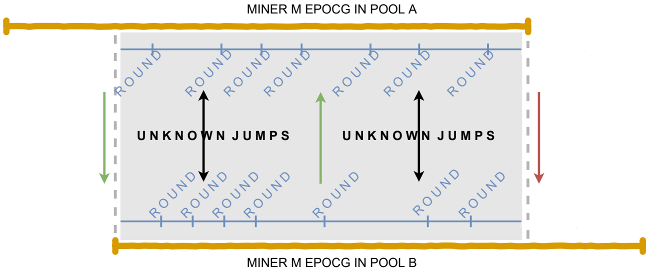


Figure 9: Windowing representation and rounds inside it.

pools at will until it reached the threshold of the first. We therefore refer to these ones as *cross-epoch hoppers*. When a miner finds itself participating in two epochs at the same time, it means that it is shifting its computing power from one pool to the other to take advantage of changes in difficulty and rewards, thus compromising the pool in which it was working.

Given two overlapping epochs, the baseline jumps for hopping activity can be identified with certainty. We define 'known jumps' the jumps that we can be certain the miner has performed, as guaranteed by the occurrence of overlapping epochs. In Figure 8 we can see that a hopper performs at least 3 jumps in the first configuration and 2 jumps in the second. Highlighted in red are the jumps made to continue in another pool immediately after receiving a reward, i.e., the jumps that end an epoch and place the miner in a different pool for the next reward. In green are those that a hopper performs to start a new epoch in a second pool, from the pool in which it has an epoch in progress. In case 1, the epoch abandoned to start another was also the first to end, indicating that the hopper, in order to reach its conclusion, returned to work from the second pool to the first one, with the intermediate jump. This jump does not occur in case 2, as the epoch started second is also the first to finish so the hopper may not have left it before reaching the threshold and returned to the first only later. In between, the hopper may have performed an undefined number of hops, which are in addition to those that are performed necessarily to work in two pools simultaneously.

7.3. Windowing

Given two overlapping epochs, we call 'window' the period of time in which a hopper jumps from one pool to another, one or multiple times. The window starts with the beginning of the second epoch and ends with the end of the first epoch to end. In Figure 9 we can see that a window is common between two epochs, but its effect is uniquely related to each epoch. The number of jumps a miner makes between two overlapping epochs is not known, nor are the number of shares it submitted and the rounds it participated in. To distinguish from the known jumps, identified by pool switching, we call 'non-observable jumps' those that a miner performs to submit shares to different pools, but without these shares allowing it to receive a reward, thus remaining unreported publicly.

The windowing feature allows us to understand three important factors of the hopping dynamics.

- First, the percentage that the window covers over the overall length of an epoch gives an idea of how profitable the jumping dynamic was for the hopper in that epoch. In the case where the percentage is high, the hopper earned most of the minimum reward threshold in the jump window but working more in the current pool. Conversely, if the percentage is low, most of the threshold is earned by fair mining outside the window. In this case, it is likely that the hopper paused mining in the current pool during the window period to start a new epoch of work in a second one. This is confirmed by looking at the percentage that the window covers on the second pool, since we expect that in the second pool most of the window covers the epoch. The threshold is reached mainly in the jump window, but still mining there. Indeed, it is likely that the hopper did not jump back and forth, but only took advantage of some variations to take a break in one pool and dynamically work in another pool in the meantime, earning an extra reward and not wasting time.

- Second, given the start and end times of the window, we can focus exactly on the rounds performed by the two pools within it. The number of rounds allows us to estimate a number of unknown jumps that the miner could have performed. Given the rewarding methods commonly used, it is likely that a hopper is only jumping on a round length basis, so the number of jumps ranges from a minimum of two to a maximum given by the average number of rounds in two pools. In this case, one could say that the number of unknown jumps is proportional to the number of rounds within the window.

- Third, by analyzing the number of rounds, it is possible to understand whether, relatively to the time window, the rounds were few and therefore lasted longer, or they were many. The insight could give some hints on which is the most advantageous configuration, therefore the one that drives the jumping dynamics. The notion about the duration of rounds inside the window allows us to study the behavior of a hopper. For this purpose we can consider the indicator:

$$\frac{window\ length}{number\ of\ rounds} \tag{1}$$

In the hypothetical case in which each validation lasts 10 minutes, the number of rounds would be directly proportional to the length of the window. The difference from the hypothetical case gives the hopper the opportunity and motivation to jump into the second pool accordingly to the duration on rounds. In particular, we call *smart* the behavior of those miners who took advantage of the dynamics, making their windows last accordingly and directing their computing power in the right direction. Conversely, we call a miner *dumb* if it shows a behavior that is inconsistent with the situation in which it performs hopping.

7.4. Detailed algorithms

We implemented a procedure that investigates the epoch and identifies hoppers. Specifically, the first procedure aims to find cross-epoch hoppers, those who jump between pools in overlapping epochs of work. The second aims to count the number of pool changes for miners who performed distance jumps, presenting no overlap between their work epochs. Finally, we implemented a procedure to analyze the windows that are created between and within two overlapping epochs, and to count how many rounds are completed by the two pools in each window. In addition, hopping windows are returned by the procedure with reference to their length over the total duration of each epoch that identifies them.

The task of algorithm 6, HOPPERS IDENTIFICATION, is to distinguish cross-epoch hoppers from intra-epoch hoppers. As input it receives the epochs set E that references all hoppers' epoch, indexed by miner and pool. In output it returns the cross-epoch hoppers set C and intra-epoch hoppers set I, resulting from the analysis of their working epochs and collected according to their distinction. First, the procedure discards with condition 6 (Table 2) the analysis on miners that present epochs always in a single pool, as they are fair miners. Second, when the condition is TRUE, being SINGLE-POOL(m, E) FALSE, each miner is assumed cross-epoch and added to C. The investigation of the miner's contribution actually begins when, for each epoch worked by the miner, the procedure checks all previous epochs to see if there are any that overlap, proving of simultaneous working. To do so epochs must be sorted with respect to their conclusion. Therefore, each of the previous epochs has the associated rewarding transaction earlier that the previous one. Given this configuration, to say that a pair of epochs, composed by the current and one of the previous, are overlapped, conditions 7 and 8 (Table 2) must be met.

The resulting epochs, positive to the concurrent work, are saved with two-by-two logic, just as they were analyzed, to facilitate their comparison in the windowing algorithm. The pairs of epochs are stored in C, indexed with the miner to which they refer. At the end of the procedure *Condition 9* applies.

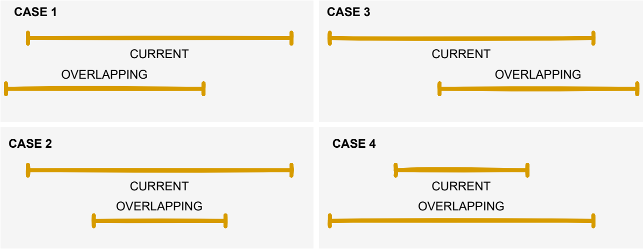


Figure 10: Possible configuration of overlapped epochs.

Algorithm 7, POOLS MIGRATION, receives as input I resulting from algorithm 6. The purpose of this procedure is to identify how many pool switches are executed by this type of miners, thus whether there is hopping behavior that is as active as the simultaneous mining. To do so, the procedure accesses all rewards received by each hopper in I, sorted chronologically. A change of pools is controlled by the 'Increment rule', which returns TRUE if two consecutive rewards are received from different pools. Miners in I are associated with a counter for each pool from which each of them receives rewards. When a miner receives a reward from a different pool than the previous one, its counter for the previous pool is incremented, indicating that there has been a jump from the pool of the previous reward to the pool of the next one. I is returned in the output of the algorithm updated with the number of jumps that each intra-epoch hopper makes from the pools.

Finally, the algorithm 8, WINDOWING, aims to represent the phenomenon of windowing between two overlapping epochs, and, for each epoch that makes a window, assigns the rounds that the epoch pool completed in the overlapping interval. The algorithm receives as input U and C. Given two simultaneous epochs, there are four configurations in which they can be arranged. With conditions 10 through 13 (Table 2), the algorithm sets the start and end of the window according to the current configuration of the couple of epochs. In Figure 10 the possible configurations of a pair of epochs are shown graphically.

Once the overlap interval is calculated and the start and end of the window are set, the algorithm searches the schedule of rounds for all the validated rounds in the interval. The overlap window is fixed for both epochs of the pair, but each is placed in a different pool, so the rounds are searched individually for each pool. In addition to the rounds that each pool has validated in the interval, the window is also assigned the percentage that it covers over the length of each of the epochs; for each analyzed window, it is saved singularly, once for each epoch of the pair, reporting the percentage covered over the epoch and the laps that the pool has validated in the interval. All window references are stored in the windows set W, sorted by miner, which is returned as output by the algorithm.

Appendix D gives anumerical example of the algorithms execution.

Algorithm 6 Hoppers identification

```
Input: epochs set $E$
Output: cross-epoch hoppers $C$, intra-epoch hoppers $I$
for $m \in E$ do
    if SINGLEPOOL($m$, $E$) is FALSE then                ▷ Condition 6
        $C \leftarrow m$
        $\bar{E} \leftarrow$ CONSEPOCHS($m$, $E$)
        for $e1 \in \bar{E}$ do
            $d \leftarrow$ INDEX($e1$)
            for $n = 1 \rightarrow d$ do
                $e2 \leftarrow$ GETEPOCH($m$, $n$, $\bar{E}$)
                if DIFFERENTPOOL($e1$, $e2$) then         ▷ Condition 7
                    if $e1[start] \leq e2[start]$ then    ▷ Condition 8
                        $C[m] \leftarrow [e1, e2]$
                    end if
                end if
            end for
        end for
        if $C[m] == \emptyset$ then                        ▷ Condition 9
            delete $C[m]$
            $I \leftarrow m$
        end if
    end if
end for
```

Algorithm 7 Pools migration

```
Input: intra-epoch hoppers $I$, rewarding set $R$
Output: intra-epoch hoppers $I$
for $i \in I$ do
    $T \leftarrow$ GETORDRDRWTS($i$, $R$)
    for $t \in T$ do
        if SAMEPOOL($t$, $t+1$) is FALSE then    ▷ Increment rule
            $I \leftarrow$ INCREMENT($i$, $t[pool]$)
        end if
    end for
end for
```

Algorithm 8 Windowing

```
Input: rounds set $U$, cross-epoch hoppers $C$
Output: windows set $W$
for $c \in C$ do
    for $couple \in c$ do
        $e1 \leftarrow couple_1$
        $e2 \leftarrow couple_2$
        if $e1[start] \lesssim e2[start]$ then        ▷ Condition 10
            if $e1[end] \lesssim e2[end]$ then        ▷ Condition 11
                $start \leftarrow e2[start]$
                $end \leftarrow e1[end]$
            else
                $start \leftarrow e2[start]$
                $end \leftarrow e2[end]$
            end if
        else if $e1[start] \gtrsim e2[start]$ then    ▷ Condition 12
            if $e1[end] \gtrsim e2[end]$ then         ▷ Condition 13
                $start \leftarrow e1[start]$
                $end \leftarrow e2[end]$
            else
                $start \leftarrow e1[start]$
                $end \leftarrow e1[end]$
            end if
        end if
        for $e \in couple$ do
            $u \leftarrow$ GETROUNDS($strat$, $end$, $e[pool]$, $U$)
            $l \leftarrow e[end] - e[start]$
            $percentage \leftarrow (100/l)(end - start)$
            $w \leftarrow (start, end, e[pool], percentage, u)$
            $W \leftarrow w$
        end for
    end for
end for
```

8. Measurements Analysis

In this section we analyze the numerical results of our pool-hopper detection framework. The main figures are the composition of the miners population and the measurement of how many among them perform hopping, and how and with which rewarding performance. The Python code of the implemented algorithms is made available in [6].

8.1. Bitcoin network setting

Our measurement campaign covers two 3-months period: the first starting on May 1, 2020, and the second starting on May 1, 2021. The bitcoin network has largely increased in size over the last few years. In August 2020, it reached 555 million transactions, and in Nov. 2021, 692 millions. With respect to the period analyzed in 2017 [3], during the 2020 period 28 million Bitcoin transactions were counted, 4 million more than 2017.

We perform the detection on the most active pools, specifically considering the five ones with the highest hash rate in 2020 indicated in Table 3. The table also reports the pools rewarding strategy, fee, number of blocks and hashrate. In particular, the pools use PPS and its variants as rewarding methods, except the Ant Pool which also implements PPLNS. All fees are under 5%, attracting new miners and hopping miners.

Mining pools	Rewarding methods	Fee	Blocks	Hash rate
Ant	PPS, PPS+, PPLNS	0%	2.130	10.5%
BTC	PPS, FPPS	1.5%	10.704	10.5%
F2	PPS+	2.5%	19.616	12.9%
Huobi	FPPS	0.8%	7.566	11.4%
Poolin	PPS, FPPS	2%-4%	3.192	13.6%

Table 3
Information on the selected mining pools (May 1, 2020).

8.2. Step 1 (revenue stream)

In the first step, the flow of revenues is analyzed looking at rewarding transaction (RWTs). The procedure associates the ownership of each RWT to a pool, tracking the paths of the coins generated by each coinbase transaction.

Following the reading of the ledger, and considering the theoretical intervals related to the characteristics of a RWT, the procedure identifies the number of transactions indicated in the first column of Table 6: comparing 2020 to 2021,

Association	RWTs	
	2020	2021
To pools of interest	5.123	21.410
To other pools	2.180	18.595
Not associated to pools	4.105	20.070

Table 4
Result from the RWTs association procedure.

while for some pools (Ant, Huobi), this number stayed rather stable, for others it underwent major increase (F2, BTC) or decrease (Poolin). While for the former the increase of RWTs is correlated with the increase of the number of miners, it is not for Poolin, which suggests the change of policy in setting the RWT recipient set.

Note that the RWT-to-pool association process results in high spatial complexity; the stream originated in each coinbase transaction is constructed by searching for subsequent transactions in a limited-size set of transactions, in the 7 days (approximately) following the day of the coinbase transaction. Because of live memory limitation, we limited the size of the set of transactions to 2 million transactions; with this limit, 30% of the RWTs identified as such are not associated with a pool, as reported in Table 4. This limits the characterization of remunerations that are far in time from the generation of the coins that fund them.

Figure 11 and Figure 12 show, respectively, the distribution of the amount of BTC sent for RWTs, and of the number of transfers of RWTs[1]. Looking at the median values of the distributions for each pool, we can see that the payment dynamics significantly differ among pools in 2020, and are similar in 2021. While in 2020 in many pools the profitable transactions contain many payouts, in 2021 this does not seem to happen. One reason can be the larger scale of the dataset for 2021 than for 2020 with therefore a thinner distribution around the average, or an important alignment of pools to each other in the year elapsed from 2020 to 2021.

In 2020, the range of each payout is also different from one pool to another. We can see that pools that send transactions more frequently tend to send smaller transactions as a general trend, while it is the opposite for pools that send transactions less frequently. For example, one can notice from Figure 12 that Ant Pool almost always sends around 100 payouts, of smaller amounts than other pools. Poolin Pool also has a near constant number of payments, around 500, but sends far fewer transactions, hence increasing the amount of each payment. This specific behavior for Poolin Pool can also be observed in 2021 (yet with a higher variance on the number of transfers), but with a lower payment amount gap with respect to the other pools.

8.3. Results from miners identification

Let us present the results from the miners identification procedure. As explained in the section 2.1, it is not possible to get an exact miners identification; indeed, a miner may choose not to reuse the same addresses, to never be identified as the author of two distinct mining actions. To tackle this problem we use a heuristic to estimate a number of miners that is as close as possible to reality. Given this estimate, we can evaluate what aspects of uncertainty it results from, so that we can tell whether our result is an approximation by excess or by default. Recalling Section 5, the result obtained by the adopted heuristic is an upper bound since it could happen that two groups of addresses are controlled by the same miner but do not have any address in common, so they are not joined (transitive property). On the contrary, in the case there is some transactions resulting from the mixing of services that is not recognized by the procedure as coinJoin, it would result in the joining in the same group of addresses of two different miners. This would increase the number of miners, so the effect of the coinJoin filtering strategy is a lower bound.

Mining addresses	2020	2021
All	261.191	2.302.434
Used in many transactions	204.172	1.915.519
Used in one single transaction	57.019	386.915
Non-singleton miner address clusters (candidate hoppers)	75.292	430.392

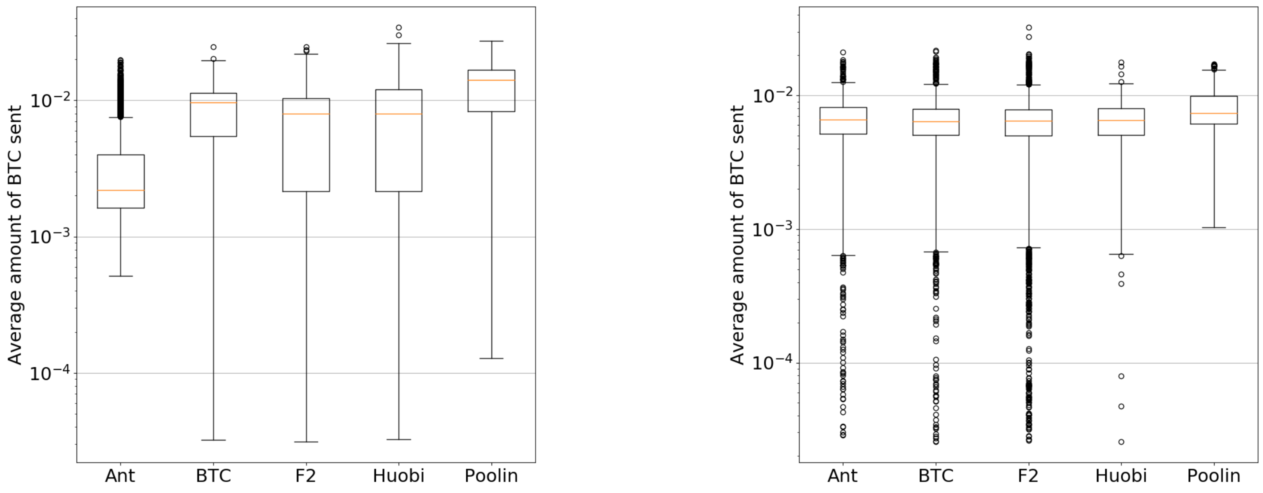
Table 5
Result of the miners identification procedure. The number of unique miners belong or do not belong to a pool.

Our miners identification lies in between the two approximations, one upward, given by the limits of the heuristic, and one downward, given by the limits of the coinJoin filter. Following this approximation, we obtain the figures in Table 5; we can see that the system, in the measured periods of 2020 and 2021, has roughly 75 and 430 thousand active miners that are candidate hoppers, respectively, i.e; those corresponding to non-singleton clusters of the heuristic (those in the singleton clusters can still be considered as miners, but the likelihood that they are also hoppers is reasonably null[2]. Empirical results from Table 5 show that between 15% and 20% of the addresses receiving a reward are used only once to receive a RWT: they could belong to already identified miners (using the heuristic), or to other miners who did not send transactions in the period. The last line of Table 5 indicates the clusters of addresses with more than one address: these are candidate hoppers (remembering that those that fall in singleton clusters are likely not to be hoppers).

In the Table 6 (Miners columns) one can see the number of miners populating each pool. As we recall, a miner can actually be present in multiple pools. It is worth noting that the average number of addresses per candidate-hopper miner, estimated as the ratio between the number of miners (non-singleton clusters, in Table **??**) to the number of RWTs (Table 6) is rather constant in time and around 20.

[1] for these and following plots, a boxplot format is used, showing the minimum, first quartile, median, third quartile, maximum and outliers

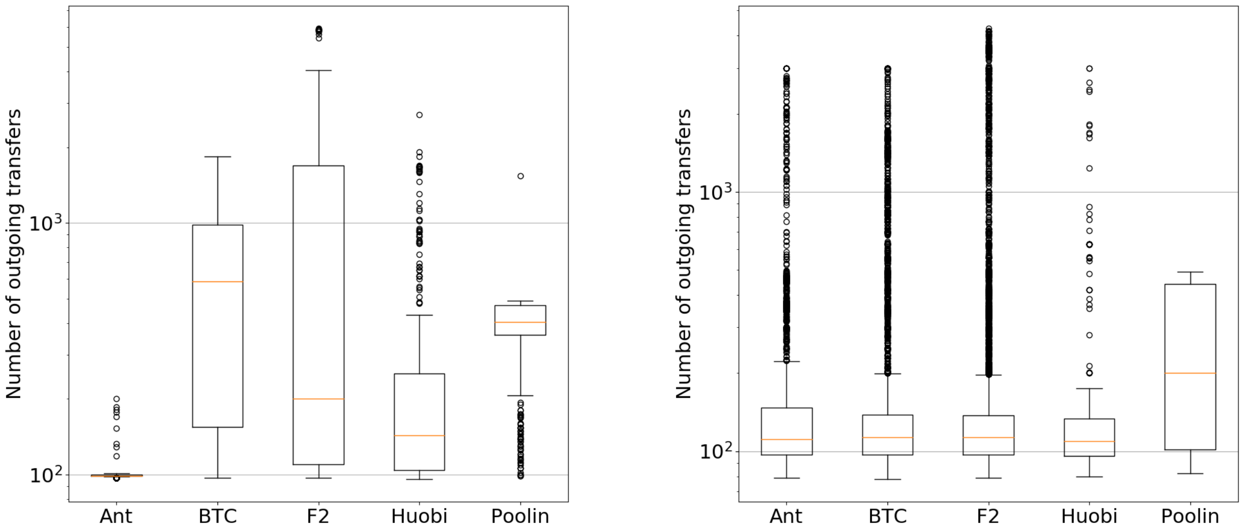
[2] Indeed, as hoppers do not want to be identified as such by the pools the mine for, they would certainly use different addresses and not the same one.



(a) 2020

(b) 2021

Figure 11: Distributions composed of the average amount sent by a single transfer of a rewarding transaction.



(a) 2020

(b) 2021

Figure 12: Distribution composed of the number of outgoing transfers of a rewarding transactions.

8.4. Results from epoch establishment

The procedure for defining epochs reveals that the median epoch length (period between two consecutive rewards) has decreased by roughly a factor 3 from 2020 and 2021. Figure 13 shows it was approximately 25 hours in 2020 and 30 minutes in 2021, with however an important number of high value outliers.

We also apply an adjustment, marked as 'after' in Figure 13: we consider continuous work intervals to be those that reside within the 2020 third quantile (e.g. 75 h and 25h, respectively, in 2020 and 2021); that is, we use it as the limit for determining the duration of an epoch as realistic or not. Indeed, beyond this limit, a common policy is that the miner stops for a while upon reward reception, changing mining pool in case of a hopper. Looking at the per-pool epoch length after adjustment in Figure 14, we see that on some pools paid in 2020 miners less frequently, hence with longer epochs, which can reduce the accuracy of placing miners in their epochs: our approximation places them with certainty in the closest pay period, although they may have worked even longer than the limit; other pools, however, straddle the limit epoch, so we consider their approximation accurate. No relevant differences can, however, be spot for 2021; a behavioral alignment seems to have happened among pools over the observed years.

8.5. Results on hoppers detection

From Table 6 (General Hoppers column) the actual hoppers among the candidate hoppers (miners with multiple addresses) are close to the half of the miners population, with a range from 13% to 74% depending on the pool and the year. The phenomenon is, in our opinion, surprisingly spread among miners.

8.5.1. General migrations

To understand the level of dynamism that is possible to pursue in each pool, we considered the 'desertation rate ', computed as the ratio between the number of outward jumps and the number of pool hoppers.

Figure 26 shows the at which extent pool is left upon a reward, to chase a second reward in another pool; a strong increase happens between 2020 and 2021, which may correspond with the usage of more sophisticated hopping strategies. Ant pool is particularly impacted by desertation, in both years, even if the percentage of hoppers strongly differs from 2020 to 2021; one reason may be the null fee for Antpool. This shows the level of dynamism of users between completing rounds, but not during rounds. Therefore, the pool configuration is allowing hoppers to carry a very active dynamic-and-receive intertwined rewards. Conversely, in pools where the rate is lower it means that fewer hoppers were able to implement a jumping strategy between rounds. Another possibility is that changes in pool configuration have made it attractive to jump only at certain times. BTC Pool and Poolin Pool, for example, have a similar length of working epochs in 2020, so there is no disparity in the frequency of rewards, but at the same time the latter is much more dynamic than the former.

Before discussing the division of hoppers into the two types of behavior, it is pertinent to look at the most common migrations across pools. A pairing is measured based on the number of consecutive rewards from different pools. Migrations are compared in relation to the overall population of hoppers and not the one in each pool; the precise reason is that each pool has a different percentage of hoppers. What we are interested in is, going beyond the number of hoppers in each pool, understanding which is the most traveled pool combination. From Figure 16 one can see that migrations standing out the most occur with the BTC Pool, in both years. A particular pairing exist between BTC and F2; the reasons are not clear, but the phenomenon accentuated significantly, both in absolute and relative values, in 2021.

8.5.2. Hoppers categories

Let us know look at the differences between 'intra-epoch' hoppers - which change pools but conclude their work epochs before jumping - and 'cross-epoch' ones - that instead exhibit simultaneous work.

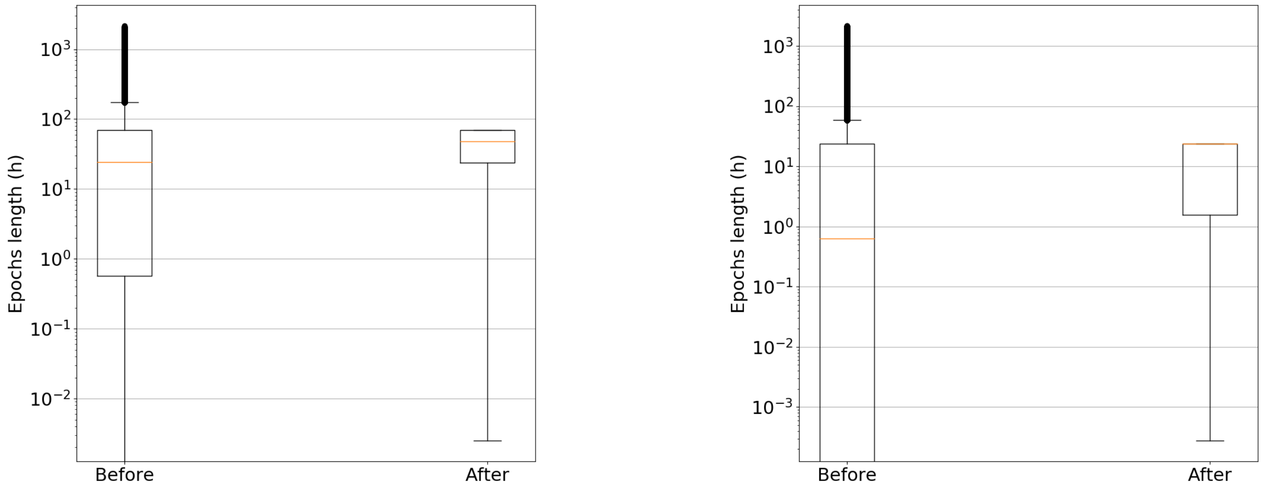
Table 6 (right side) shows how the behavior distribution differs for different pools over the two considered. We see that, in both years, roughly half of the hoppers are cross-epoch hoppers and half are intra-epoch hoppers. For some pools the difference is more pronounced in 2020 than in 2021, when an close-to-even share is observed.

The same distinction is also reported in the epoch duration measurement - see Figure 14 for 2020 only. In fact, cross-epoch hopping behavior is likely to be slightly better executed in pools with faster reward distribution. The peculiar dynamics of Ant Pool, for example, disfavors static work, inducing miners to jump between epochs. In contrast, while mining in F2 Pool and Huobi pools, simultaneous work in the other three pools is feasible but not trivial.

For the specific case of 2020 showing more differences among pools, in Figure 17, we can see the pools that have the highest percentage of epochs performed simultaneously with epochs in other pools. Ant Pool appears as the most suitable for doing simultaneous working, as 50% of the work epochs performed in it are overlapped to epochs in other pools. What is interesting to note is that miners actively jump into both pools that allow them to reach rewards faster and pools that force them to work longer to finish an epoch. For example, Huobi Pool imposes a longer time to reach rewards, but similarly a high percentage of epochs are overlapped with epochs in other pools. Although not the preferred behavior of most miners in this pool, it makes us realizing that it is strategically advantageous for a hopper to jump into pools where epochs are concluded more dynamically while in the middle of a pooled epoch where reward attainment is slower.

Another aspect to consider is the percentage of overlapping windows over the total length of epochs, shown in Figure 18. We have already presented in which pool dynamic mining is preferred, so here we can understand where it actually helps to achieve the reward and, instead, where jumping elsewhere involves the interruption of the validation in the current pool.

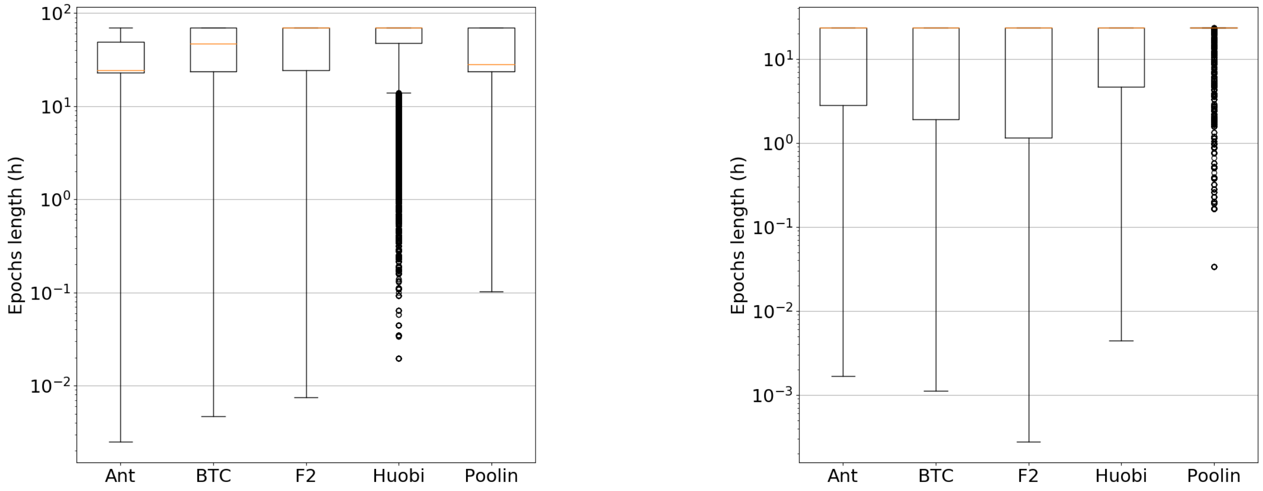
For 2020, in Ant Pool, the percentage of dynamic work during an epoch is 80%, meaning that the rewards are almost entirely in epochs overlapping with others; in contrast, the percentage of overlaps in Poolin Pool epochs is notably smaller than the total length of the epoch: the epochs in this pool, which is one with the prevalence of jumps between epochs, are often paused to mine elsewhere and then restarted. A realistic case, for example, might see a hopper mine in Poolin Pool, then pause the current epoch to jump into Pool Ant, where it completes a few epochs, and then go back at will to continue validation in Poolin Pool. By doing so, the miner efficiently shifts its resources to get a few small rewards from Ant Pool and a larger one from Poolin Pool, instead of just one large one from Poolin Pool a little



(a) 2020

(b) 2021

Figure 13: Distribution of epochs lengths before and after adjustment.



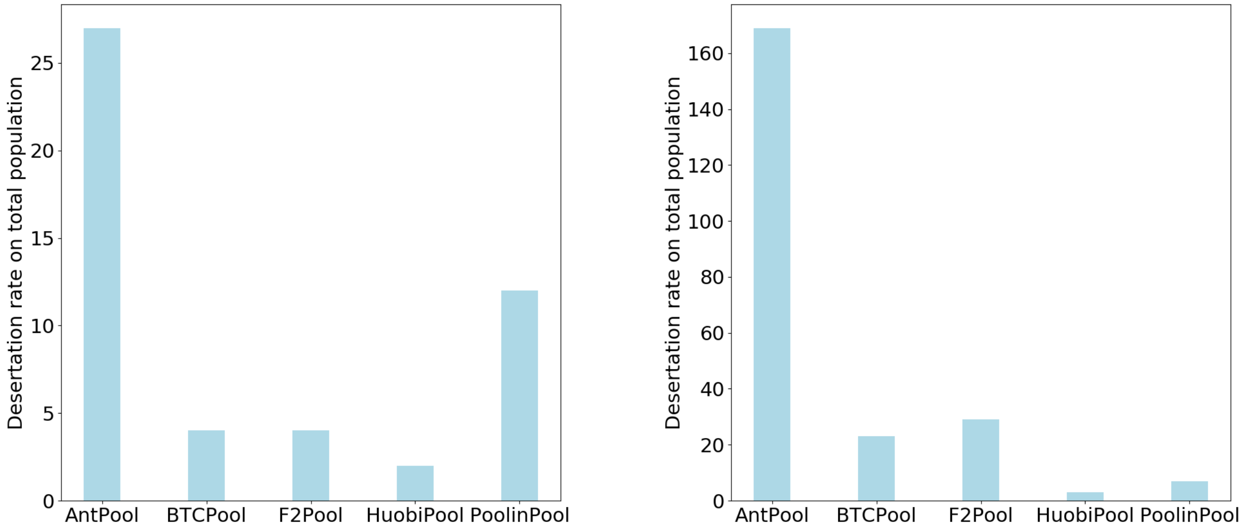
(a) 2020

(b) 2021

Figure 14: Distribution of epochs lengths in each pool.

	RWTs		Miners		General hoppers				Cross-epoch hoppers				Intra-epoch hoppers			
					Total		%-miners		Total		%-general		Total		%-general	
Pools:	2020	2021	2020	2021	2020	2021	2020	2021	2020	2021	2020	2021	2020	2021	2020	2021
Ant	3.622	3.485	8.091	84.847	1.045	34.575	13%	41%	778	16.477	74%	48%	267	18.098	26%	52%
BTC	702	5.911	29.066	147.188	11.129	48.857	38%	33%	3.594	22.985	32%	47%	7.535	25.872	68%	53%
F2	458	11.469	33.555	268.678	11.488	58.711	34%	22%	3.532	27.078	31%	46%	7.956	31.633	69%	54%
Huobi	423	458	23.688	16.353	10.881	9.555	46%	58%	3.805	5.005	35%	52%	7.076	4.550	65%	48%
Poolin	288	87	3.970	3763	1.610	2.772	41%	74%	983	1.080	61%	39%	627	1.692	39%	51%

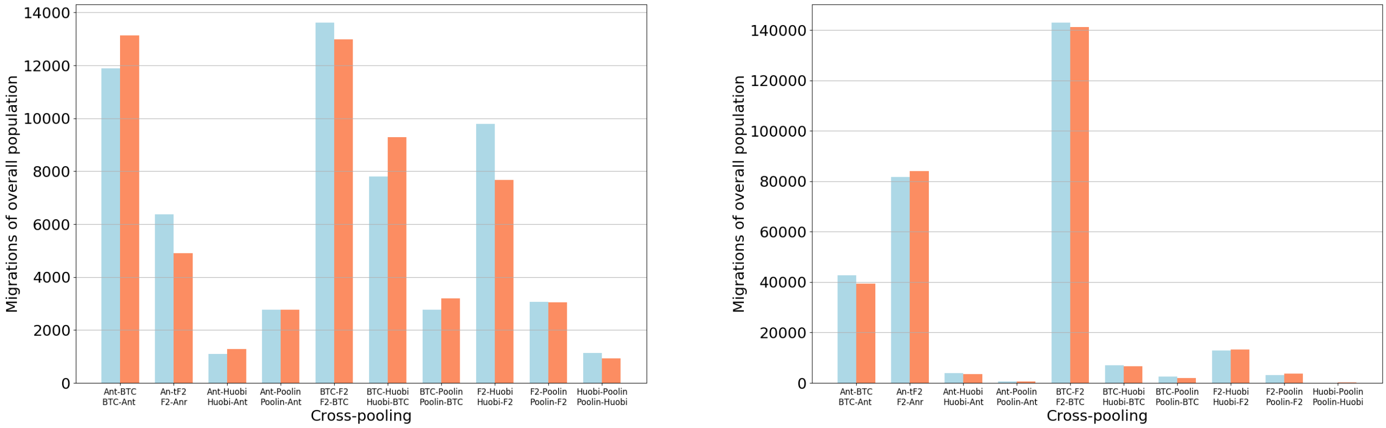
Table 6
Characterization of the RWTs, miners and hoppers for five selected pools.



(a) 2020

(b) 2021

Figure 15: Desertation rate in each pool.



(a) 2020

(b) 2021

Figure 16: Preferred pool migrations.

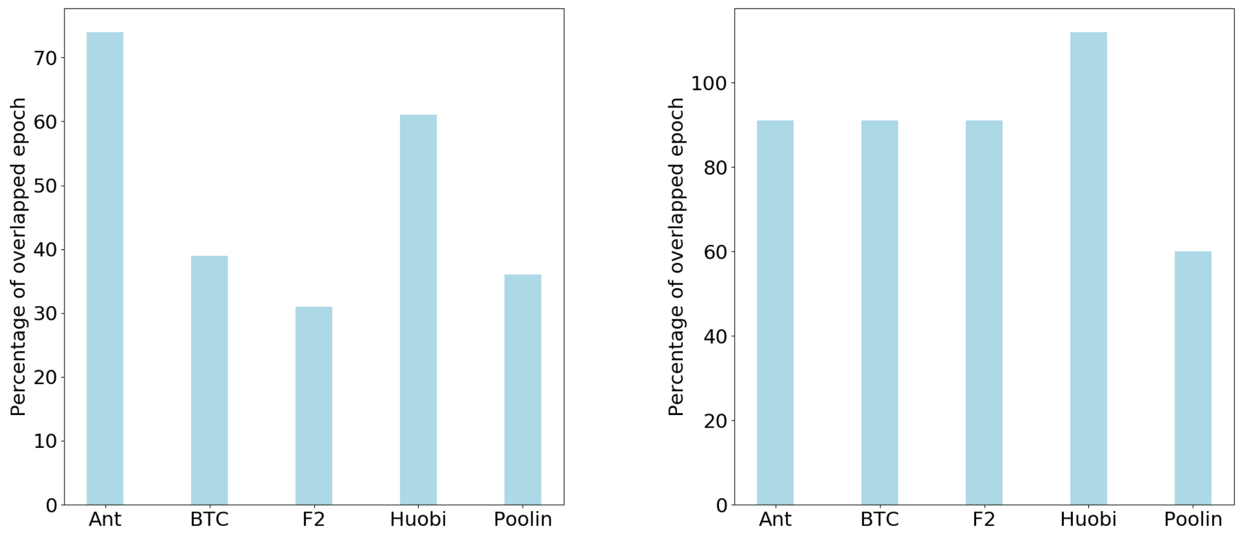
faster. This general trend can be also observed in 2021.

The Figure 19 shows the percentages of a round's duration over the entire simultaneous work period (resulting from (1)). In other words, it reveals in which pools the rounds are validated the fastest while the miners are actively jumping. I For the Huobi Pool, for both 2020 and 2021, when hoppers are working in simultaneous epochs: the validations are won much more slowly than in other pools. Hopping in a pool to contribute to slower rounds does not seem very efficient, which might explain why in Huobi Pool intra-epoch hopping is preferred. In this respect, the situation changes for the Poolin pool from 2020 to 2021. In contrast, in the other pools the validations are quicker, even though miners jump, allowing for faster rewards. It is certainly efficient to participate in these pools and be part of a quick validation, in case a miner is willing to interval their work in a pool with longer time frames to achieve the rewards.

8.5.3. Remuneration performance

To conclude, we analyse the extent to which the phenomenon is more profitable for hoppers than for stable miners, looking at Figures 20 and 22. We note that:

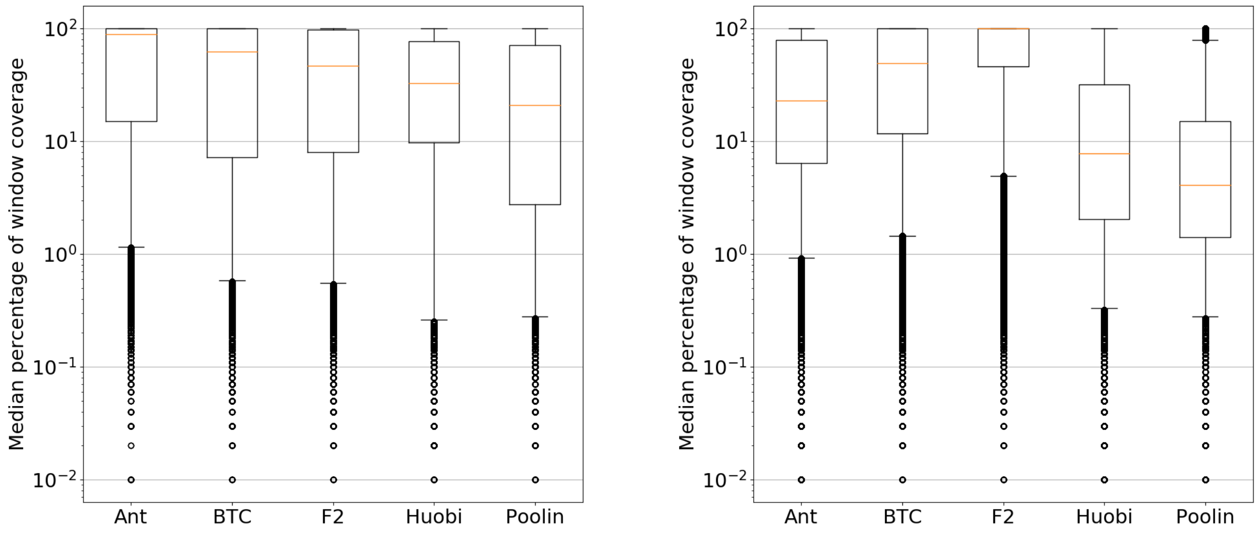
- as shown in Figure 20 (we add to the boxplot the av-



(a) 2020

(b) 2021

Figure 17: Percentage of epoch carried out by cross-epoch hoppers.



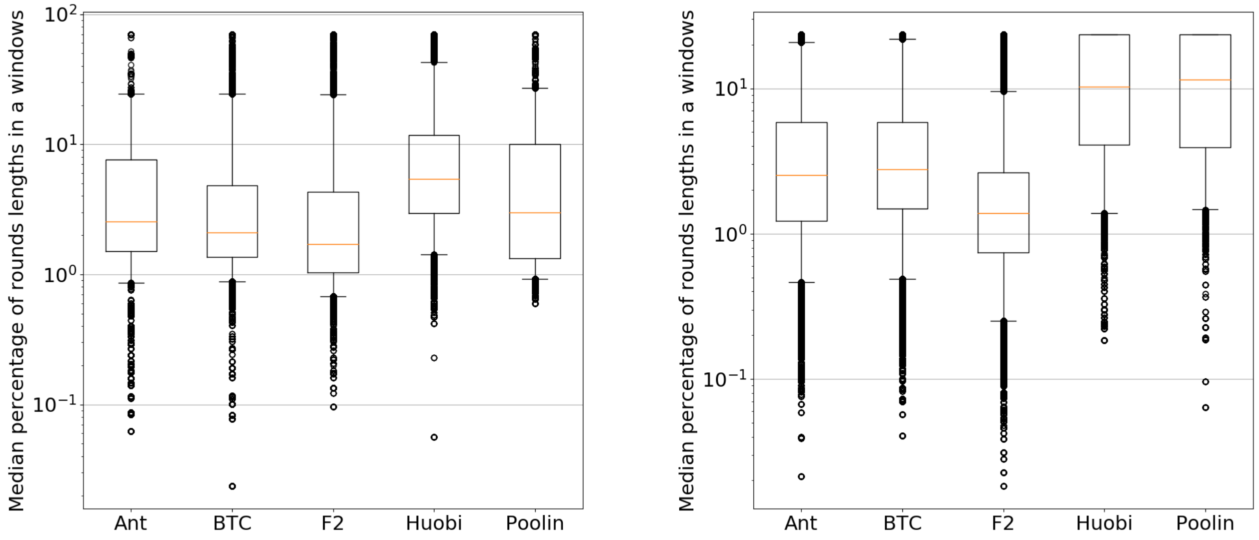
(a) 2020

(b) 2021

Figure 18: Distribution showing the coverage values of windows over the epochs.

erage as a red asterisk for 2020 as they do visibly differ, likely because of the smaller population), for all three categories, non-hopper miners (counting only those miners with more than one address), intra-epoch hoppers and cross-epoch hoppers, the average gain *per transaction* over the period is statistically the same in both years. Indeed, this is a direct result of the reward methods implemented by the pools to prevent the phenomenon from being unfair to other miners.

Unlike when proportional rewarding methods were used (whereby the distribution of rewards rewarded hoppers doing early mining to a greater extent), now every miner is rewarded at a roughly same amount for the work it does, regardless of when it is done and whether



(a) 2020

(b) 2021

Figure 19: Distribution showing the length of rounds in each window.

it is a hopper or not. The current rewarding methods therefore prevent that there is a higher average gain for hoppers for equal work. Specifically, rewards range from 0.002 BTC, for rewards generated as a result of PPS while they are higher up to 0.006 BTC in cases where a minimum number of shares must be submitted to achieve the reward. Therefore, the phenomenon can no longer be referred to as unfair towards anyone.

- looking at the Figure 22, no major difference can be observed between cross-epoch hoppers and intra-epoch ones. That is, pool-hoppers can exploit the hopping behaviour to gain more across pools over time. In fact, we notice that hoppers get rewards roughly 10% more rapidly than single pool miners. Furthermore, as can be seen from Figure 21, in 2020 for instance, cross-epoch hoppers receive a average reward 33% higher on median than other miners, due to the increased frequency of rewards over the period, so somehow denoting a more efficient mining strategy by obtaining more rewards. Note that this is not to be considered as an unfair increase, in our opinion, but rather an increase driven by a more effective strategy that allows miners to hop and make the most out of their resources.

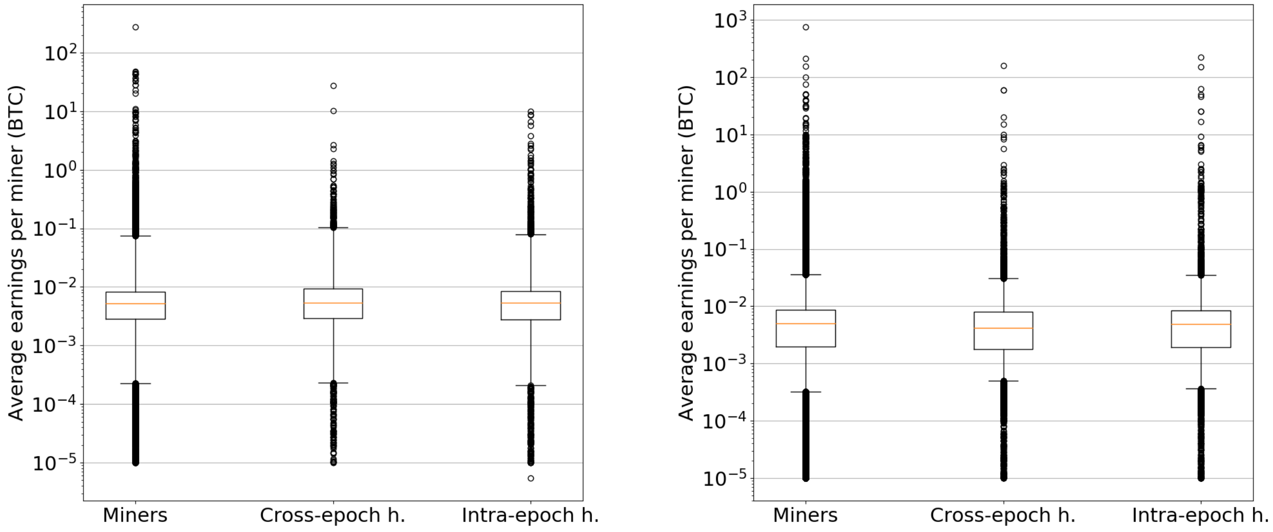
9. Positioning with respect to related works

Our algorithmic framework can be qualitatively compared to two empirical detection papers, already briefly presented in the background section. The first one implements a different algorithmic and evaluation strategy in [22]. The second one does not consider important and more newly born hypotheses [3].

9.1. Tovanich et al. [22]

Although the purpose is the same across the board and falls into the detection category of pool hopping, there are differences in the level of detection we want to achieve with the approach in [22]. In fact, their aim to measure miner migrations covering different years, resulting in a broad data statistic that characterizes cross-pooling as a function of the fee applied and the rewarding method used by pools in each year. In contrast, because fee variation has now stabilized and most pools primarily use the same rewarding method, or variations of it, we aim to estimate a precise number of active miners in a given time period and given the current conditions. Only in relation to this number we are interested in understanding what percentage of miners performs hopping, with what pool characteristics it is most performed and if indeed it is fair today.

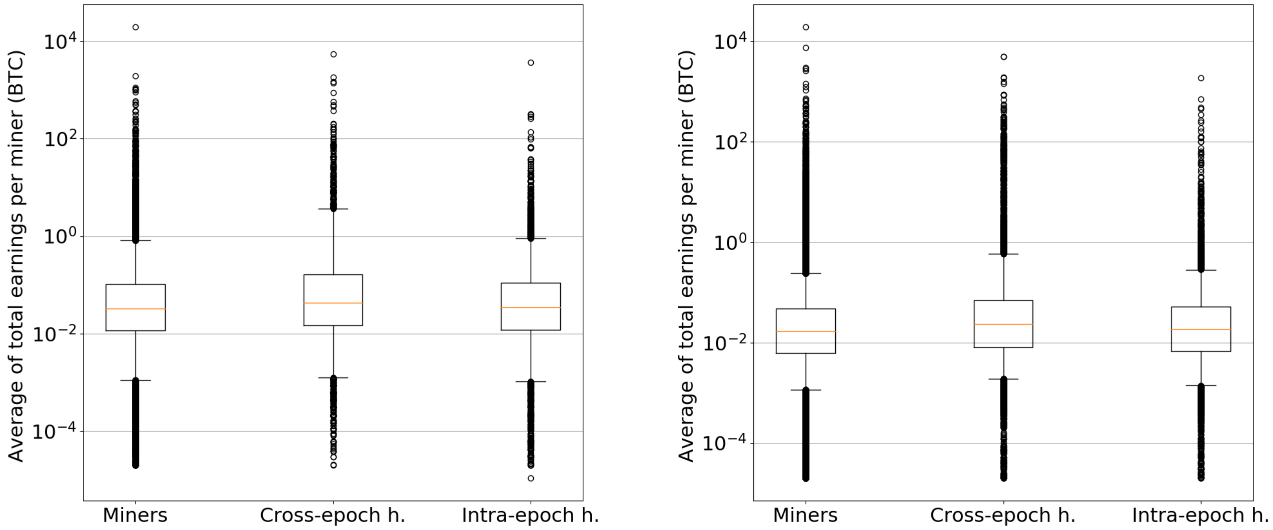
Methodologically, an important difference is the detection strategy. In [22], it relies in the association of pool ownership. In our framework, we assign the ownership of a transaction to the pool that has the least used coins within the transaction's funds. Instead, they assign ownership to the pool with the largest amount of funds, even if generated long before the transaction. With their strategy, the pool of ownership would be assigned as the one with the highest percentage of the funds funding the transaction. As we mentioned, the uncertainty of assigning a certain coin to the pool arises when the coins are the result of numerous transfers. Thus, even if a transaction has a higher percentage of funds from one pool and a lower percentage from another, but we do



(a) 2020

(b) 2021

Figure 20: Boxplot distribution of per-transaction rewards miners of static miners, cross-epoch hoppers and intra-epoch hoppers.



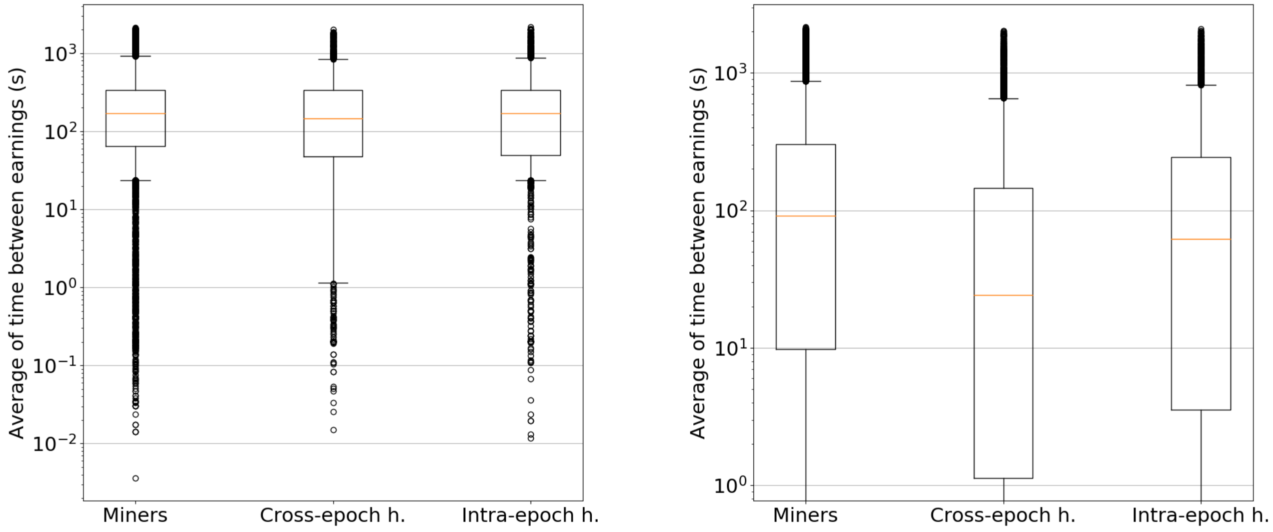
(a) 2020

(b) 2021

Figure 21: Boxplot distributions of cumulative gain earned in the period by static miners, cross-epoch hoppers and intra-epoch hoppers.

not know how far back in time those coins were generated, we still cannot be certain that the transaction was sent from one pool instead of the other. In fact, it could be that, after several transfers, a pool receives coins from outside and uses them all to pay a reward, but then does not turn out to be the owner, since the coins were generated for another one. In contrast, by focusing on the journey of a coin, a coin that has gone through fewer transactions is very likely to still belong to the pool for which it was generated.

Another main difference lies in miners identification. Because their intent is on measuring cross-pooling traffic, they focus on individual miner addresses, without knowing whether



(a) 2020

(b) 2021

Figure 22: Boxplot distribution of the time between earnings of every static miner, cross-epoch hopper and intra-epoch hopper.

actions from different addresses are associated with the same user. Furthermore, the authors argue that the heuristic [17] cannot be used because it is inaccurate and induces false positives, since, in the case where miners use mixed trades or services, the heuristic would group them in the same cluster. With respect to our measurement goal, considering individual addresses does not allow us to place miners in pools and characterize their work. In addition, they say there is a 25% in addition to the hopping rate by considering miners with groups of addresses, instead of addresses of individual miners. Our work takes a completely different direction on the issue of miner identification, since we are interested in placing a miner in each epoch despite using different addresses. In fact, we use the heuristic [17] and at the same time handle the imprecision resulting from the phenomenon of mixed services. by preventing as much as possible the association of addresses controlled by different entities to the same one. Removing all CoinJoin transactions before applying heuristics is crucial since it prevents as much as possible the association of addresses controlled by different entities to the same.

Unfortunately, the code in [22] is not open-sourced, which prevented us to do a fair quantitative comparison against newer data. Authors published only the dataset with their results, but given these differences in strategy and the different scope of their detection, it is not directly possible to do a quantitative comparison on recent data.

9.2. Belotti et al. [3]

Our work is a continuation of Belotti et al. [3], reflecting the dismissal of some assumptions and the application of new ones. Firstly, the identification of rewarding transactions in [3] is imprecise: it was considered that all and only coinbase funded transactions as such. In fact, a pool can manage bitcoins internally with several transactions and pay rewards only later, without the founds being derived directly from a coinbase [18].

Moreover, we consider the distribution of rewards in accordance with the existence of the minimum reward threshold. This implies that a detection based only on consecutive rewards is not sufficient to characterize the work of miners and place them in each pool at different times. As a result, many hopping cases are neglected by the previous strategy in [3]: detected hoppers are in the final rounds of epochs, without considering other rounds within them. As a quick numerical comparison with respect to this aspect, let us consider the hoppers e could detect in our method, looking at Table 6 (right side):

- method in [3] does not allow to detect the cross-epoch hoppers;
- method in [3] would not have marked as hoppers all the intra-epoch hoppers, but only to this hoppers observable in the final rounds of each epoch;
- overall, across the different pools, about half of the hoppers could not be detected with the method in [3].

Finally, when more than two pools are considered, hoppers may be rewarded in two non-adjacent reward transactions.

The result of the analysis determines that only five thousand miners are present and active over a total period of three months in 2017. Furthermore, over a two-week sample, hopping was clearly smaller: 43 pool-hoppers were detected in a population of 69 active miners with the two pools. In addition to being smaller because far fewer users are identified, it is also certainly smaller because many miners are not placed in the rounds in which they work. Measurements also show that hopper rewards significantly exceed those of static miners, as the median of rewards is 3 times higher for hopping miners than for static miners. The pools they analyzed used different reward methods than those used today, which explains such a difference in payoffs among categories. Consequently, their results regarding the fairness of the rewards are not comparable with ours, which on the contrary prove the fairness of the distribution of rewards thanks to the (relatively, recently adopted) current methods.

10. Summary

We presented in this article a framework for pool-hoppers detection and characterization for the Bitcoin blockchain. Our work goes beyond and improves the previous work [3] toward a scenario closer to the current reality of Bitcoin mining, considering pools payout streams and the existence of minimum reward thresholds. Indeed, since few years, new rewarding methods, PPS, PPLNS and variants, have been invented precisely to mitigate the pool-hopping phenomenon and are now widely used in the system. This study therefore assesses the actual efficiency of the new rules.

The proposed framework and related numerical analysis of empirical measurements, against five major mining pools during 3 months of 2020, allow us to determine that:

- the percentage of hoppers over the population of miners ranges from 34% to 91%, depending on the mining pools. Some pools are better for getting faster rewards, in fact they count a lot of miner migrations. In three out of five pools, cross-epoch behavior is preferred to intra-epoch behavior. This difference does not depend on the length of the epochs, as the dynamics of active jumping are best performed by moving between pools with long epochs and pools with shorter epochs.

- there is no diversity in the per-transaction rewards of static miners versus hoppers any longer, hence differently than what happened in the past with proportional rewarding methods, thanks to the new reward methods we described.

- Cross-epoch hoppers earn more over time, 33% more on median in the observed period. Indeed, they are the ones who are able to receive rewards the fastest. The frequency of their rewards is higher than that of static miners, showing a more efficient strategy, and thus the overall gain over the long run is higher. Thus, in the long run, active hopping is more efficient than staying in a single pool or moving only between completed epochs.

About the last two findings, it is worth noting that the mining of cross-epoch hoppers is automatically prevented from harming static miners, as the single reward remains equitable to the time and resources involved. As a result, the higher frequency of rewards is not at the expense of static miners, but is merited for more dynamic work as the exploitation of the hopper's resources is maximized. Hoppers are miners who strive to earn rewards as efficiently as possible, even if that means moving dynamically through the system and not standing still in a pool. The pool-hopping phenomenon is therefore to be interpreted as a strategic decision just like the one that led to the creation of mining pools. Mining pools were created to allow miners to share their resources in order to get more stable and closer rewards over time. This is exactly the goal of those who move from one pool to another: to maximize the use of their resources and get rewards faster.

For the sake of reproducibility, we open source the code in [6]. Future work can consists in applying the analysis can be applied to another cryptocurrency with a comparable consensus method, such as Ethereum.

The adoption of novel techniques to further control, ideally avoiding pool-hopping is possible in the coming years, as for instance the frameworks proposed in [21, 7]. Our study is therefore susceptible to be repeated against possible future new settings of the Bitcoin rewarding strategies.

A. Revenue Stream Association Example

Hereafter is an example of the functioning of the first phase of our detection. Algorithms 1, 2 and 3 are considered.

Suppose we are reading the ledger and at a certain moment are shown transactions in the Table 7. Starting with TX1, to access the subsequent transactions it is not enough to know that ADD2, ADD3 and ADD4 are receiving it, because these addresses may have also sent transactions that are not consecutive to it. To find which transaction follows the current one, we need to access the ID field of all transactions by looking for IDs that match this transaction hash. What we know so far is that there are 3 transactions following TX1, so we can build up to 3 branches from here. Recall that when accessing a transaction, the ID field refers to the previous transaction, so looking at transaction 2, its backward funding shows transaction 1. Transaction 3, on the other hand, is funded by transactions 1 and 2 while transaction 4 is funded by transactions 1, 2, 3.

Matching structure D, Table 3, is constructed so that we can follow transactions onwards, since it is the result of searching for all transactions that have an ID matching the hash of a given transaction. Transaction 1 is followed by transaction 2, 3, and 4. Transaction 2 is followed by transaction 3 and 4, and so on.

Suppose we start with a coinbase transaction C1, received from ADD1 and with whose funds ADD1 sends transaction TX1. To follow up on subsequent transactions, D holds elements referencing those transactions consecutive to TX1. Table 9 shows that the referencing elements are ETX2, ETX3 and ETX4. Note that transaction 1 sends a transfer to ADD2

	in adds	hash	(IDs, pos)	out adds
CX1	-	HC1	-	ADD1
TX1	ADD1	HTX1	(HC1, -)	ADD2 ADD3 ADD4
TX2	ADD2	HTX2	(HTX1,1)	ADD3 ADD4
TX3	ADD3	HTX3	(HTX1,2) (HTX2,1)	ADD4
TX4	ADD4	HTX4	(HTX1,3) (HTX2,3) (HTX3,1)	ADD70

Table 7
Example of some transactions in the ledger.

Keys	Subsequent Transaction Elements
HC1	ETX1
HTX1	ETX2 - ETX3 - ETX4
HTX2	ETX3 - ETX4
HTX3	ETX4
HTX4	ETX70

Table 8
Example of *Matching Structure D*, linking each transaction to the next ones.

ETX2	Att.
hash	HTX2
leaf	1
od	[0, 1]
isR	False
pr	1

ETX3	Att.
hash	HTX3
leaf	1
od	[1]
isR	False
pr	2

ETX4	Att.
hash	HTX4
leaf	1
od	[0]
isR	False
pr	3

Table 9
Content of the elements referenced by each key in *D*.

ETX1	Attributes
hash	HTX1
leaf	0
od	[0, 1, 0]
isR	False
pr	1

Table 10
Content of the elements referencing the transaction currently visited.

ETX3	Attributes
hash	HTX3
leaf	2
od	[1]
isR	False
pr	1

ETX4	Attributes
hash	HTX4
leaf	2
od	[0]
isR	False
pr	2

Table 11
Content of the elements referenced by each key in *D*.

ETX2	Attributes
hash	HTX2
leaf	1
od	[0, 1]
isR	False
pr	1

Table 12
Content of the elements referencing the transaction currently visited.

ETX4	Attributes
hash	HTX4
leaf	2
od	[0]
isR	False
pr	1

Table 13
Content of the elements referenced by each key in *D*.

in position 1, to ADD3 in position 2, and to ADD4 in position 3. In *D*, under HTX1, ETX2 has the *pr* attribute reporting that the funds in the previous transaction (TX1) are in position 1. The same for ETX3 to transaction 3, whose *pr* shows position two, and for ETX4 to transaction 4, whose *pr* in turn shows position 3.

When visiting TX1, the procedure will check which outgoing transfers are to be followed and which are not. The *od* attribute in ETX1, Table 10, indicates that the first and third outgoing transfers are to be followed, while the second is not. This means that transaction two and transaction four received an amount of bitcoin from transaction one that is above the payout threshold. They are considered transfers made within the pool and not to external users, which instead is the case for the second transfer to transaction three. When HTX1 is accessed in *D*, the elements that refer to the next transaction are unordered, so to know which of them corresponds to the second transfer out of TX1, we need to access the *pr* attribute. The element showing position 2 in *pr* is ETX3, so the transaction referenced by this element is not to be followed. Elements that in *pr* show a position that in *od* has zero, instead reference transactions to be followed.

ETX2 and ETX4 identify as the next transactions to be follow, that are transaction 2 and 4. In *D* HTX2 refers to ETX3 and ETX4, Table 11.

Going down one level in the stream visit, ETX3 shows position 1 in *pr* of ETX2, Table 12. At that position *od* indicates a 0, so a branch to transaction 3 is created from transaction 2. Instead, ETX4 takes position 2 in *pr* of ETX2. At that position *od* has indicated a 1, so no branch to transaction 4 is created from transaction 2.

The process continues by doing the same for HTX3, which refers only to ETX4 since transaction 3 has a single output. Table 13 shows that the *pr* attribute of transaction 4 is 1, but at that location the *od* attribute of transaction 3 indicates that a branch from there will not be created, Table 14. The algorithm proceeds consistently until the end of the ledger. Figure 23 shows the final flow for our example.

ETX3	Attributes
hash	HTX3
leaf	2
od	[1]
ISR	FALSE
pr	1

Table 14
Content of the elements referencing the transaction currently visited.

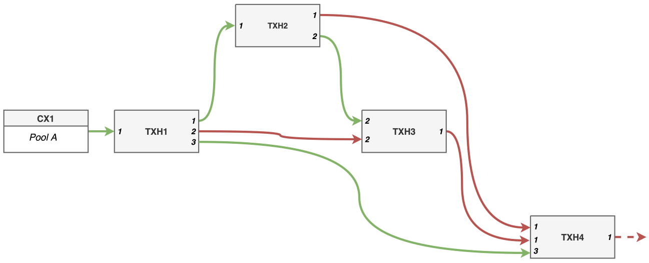


Figure 23: Figure shows the stream of transactions in subject.

	visits
TX20	(Pool P,10) (Pool R,3)

Table 15
Table shows an output example of the second algorithm, which returns all the visits of transactions.

When a transaction is accessed the leaf is updated. Transactions are only visited during a shorter path than the one in which they have already been encountered. The information about the pool that received the coinbase transaction generating the stream currently visited is also stored. At the end of visiting all streams, the rewarding transactions report all the times they were visited. As described in the previous section, the transaction property is assigned to the pool that presents it in one of its streams that is the shortest for the transaction.

Now assume that transaction TX20 is visited by tracking the stream of a pool P as a result of 10 other transfers; TX20 is then visited by tracking the revenue stream of another pool R as a result of 3 previous transfers, Table 15. To figure out which pool to assign ownership to, notice that it is more likely that the coins generated for R are still in R's wallet after 3 transactions, as opposed to P's coins, which may have left P's wallet before the 10th transaction, case 1 in Figure 24. The procedure assigns transaction TX20 to pool R because there is a higher probability that it owns some coins in that transaction. Instead, the coins originally created for P are likely to have become part of another pool's stream because they are at a more advanced level of use than the other coins in the same transaction.

B. Miners Identification Example

Hereafter is an example of the functioning of the first phase of our detection. Algorithms 4 is considered.

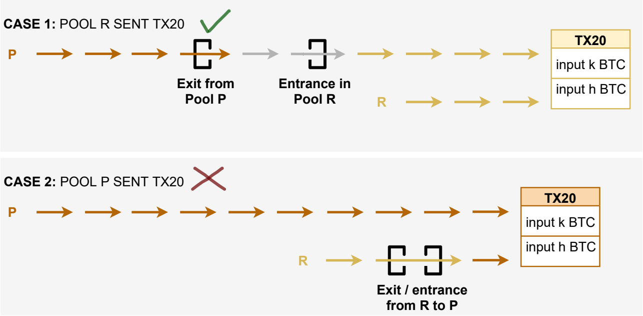


Figure 24: Figure shows the ownership of a transaction funds resulting from the revenue chain.

Address	Transactions Sent		
~~ADD1~~	HTX1	HTX2	HTX4
ADD2	HTX3		
ADD3	HTX4	HTX5	
ADD4	HTX5		
ADD5	HTX6		
ADD6	HTX3		

Table 16
Example of *Senders Set S*.

TX1 Senders
~~ADD1~~
ADD3

TX2 Senders
~~ADD1~~
ADD2

TX3 Senders
ADD2
ADD6

TX4 Senders
~~ADD1~~
ADD2

Assume to pull address ADD1 from R at the beginning of the miners identification. The address is not owned by any miner already identified, therefore the procedure can start collecting all its homonymous addresses. Accessing ADD1 in S, Table 16, we can see that it is referencing the list of hashes composed by: HTX1, HTX2, HTX4. The miner at this stage is identified only by ADD1 and it is known to have sent transaction 1, transaction 2 and transaction 4. As shown in Table 17 ADD1 is strikethrough as already visited on S and already associated to this miner, so it will not be considered anymore..

Given the three hashes, we can retrieve the corresponding transactions from T, Table 18. ADD3 is homonymous with ADD1, as they are both senders of TX1. Also ADD2 is homonymous with ADD1, as they are both senders of TX2. Therefore, even if ADD2 and ADD3 are not used in the same transaction, we can already say that they are homonymous with each others, as they have ADD1 in common.

Now that we know that also ADD2 and ADD3 are addresses controlled by the miner, let's see in S which transactions were sent with those addresses. The process gets repeated as before. From S, Table 19, we can see that ADD2 is referencing HTX3, so the miner sent transaction 3 with ADD2. Also it sent transaction 4 and transaction 5 with

TX5 Senders
ADD3
ADD4

TX6 Senders
ADD5
ADD -

Table 17
Senders addresses of the transactions in example.

TX1 Senders
~~ADD1~~
ADD3

TX2 Senders
~~ADD1~~
ADD2

TX3 Senders
~~ADD2~~
ADD6

TX4 Senders
~~ADD1~~
ADD2

TX5 Senders
~~ADD3~~
ADD4

TX6 Senders
ADD5
ADD -

Table 18
Senders addresses of the transactions in example.

ADD3. Transaction 4 is strikethrough, as it has already been accessed visiting the transaction sent by ADD1. It will not be visited again, since its sender addresses have already been associated to the miner.

The procedure, then, access again T to retrieve the sender addresses of transaction 3 and transaction 5, Table 20. Among the addresses strikethrough, already associated to the miner, transaction 3 shows ADD6 as homonym with ADD2, hence also with the other miner's addresses. In the same way, transaction 5 shows ADD4.

Again ADD4 and ADD6 are accessed in R, Table 21. Both reference already seen transactions, so we can conclude that the collection of all the homonymous addresses linked by the transitive property and controlled by the miner is completed. No transaction showed ADD5 within the inputs, so the procedure is not brought to visit transaction 6. In our example ADD5 is the only one in input to transaction 6 and, since it is not visited, ADD5 is not controlled by the miner. Table 22 shows all the addresses detected by our algorithm as controlled the by the same miner.

C. Epochs Establishment Example

Hereafter is an example of the functioning of the first phase of our detection. Algorithms 5 is considered.

Suppose we want to investigate the work performed by the miner m, so as to characterize its epochs and know its presence in the pools at any given time. The first step is to access all remunerative transactions it has received. Sorting them by time, each transaction identifies with the next a time interval by which it has reached a threshold and its work has been rewarded. As we know, it may have participated in several pools, so the pool of one remuneration may be different from the pool of the next. Figure 25 shows this situation. Accordingly, the procedure looks for all pools that paid miner m, in this case Pool A and Pool B, then proceeds to compute the work intervals in one pool at a time.

Address	Transactions Sent		
~~ADD1~~	~~HTX1~~	~~HTX2~~	~~HTX4~~
~~ADD2~~	HTX3		
~~ADD3~~	~~HTX4~~	HTX5	
ADD4	HTX5		
ADD5	HTX6		
ADD6	HTX3		

Table 19
Example of *Senders Set S*.

TX1 Senders
~~ADD1~~
ADD3

TX2 Senders
~~ADD1~~
ADD2

TX3 Senders
~~ADD2~~
ADD6

TX4 Senders
~~ADD1~~
ADD2

TX5 Senders
~~ADD3~~
ADD4

TX6 Senders
ADD5
ADD -

Table 20
Senders addresses of the transactions in example.

The Table 23 reports the work intervals of miner m. We can see that TX1 and TX2 are sent from the same Pool A and are quite close to each other. They identify two work intervals, one ending with TX1 and the other with TX2. Of the one ending with TX1 we do not know maximally how long it lasts because it is the first one, while we know that TX2 creates with TX1 an interval of only 3 hours.TX2 and TX4 are also consecutive, again from Pool A, but they identify a rather long work interval, since TX4 is 56 hours after TX2. In this case, we need to understand whether the interval between rewards is consistent with a continuous work scenario or not. In fact, it is likely that m took a break between reaching the threshold for TX2 and TX4.

To resolve the uncertainty about the work scenario, the procedure studies the distribution composed of the lengths of all time intervals between two consecutive rewards in each pool. As explained in the description of the procedure, the limit for considering a work interval as continuous is the third quantile of such distribution. Suppose this value is 35 hours.

Having computed the upper bound characterizing the work to obtain a reward with a maximum of 35 hours, the epochs of m work in the pool are established, modifying all intervals that do not fall within the bound. Consequently, we can say that the time the miner worked to receive TX4 is less than the entire interval between the two transactions. Since the end of the epoch corresponds to receiving a rewarding transaction, the epoch E3 is set as shown in Table 24. Adjustments to intervals that cannot be considered realistic epochs are high-

Address	Transactions Sent		
~~ADD1~~	~~HTX1~~	~~HTX2~~	~~HTX4~~
~~ADD2~~	~~HTX3~~		
~~ADD3~~	~~HTX4~~	~~HTX5~~	
~~ADD4~~	~~HTX5~~		
ADD5	HTX6		
~~ADD6~~	~~HTX3~~		

Table 21
Example of *Senders Set S*.

Miner
ADD1
ADD2
ADD3
ADD4
ADD6

Table 22
Miner's addresses.

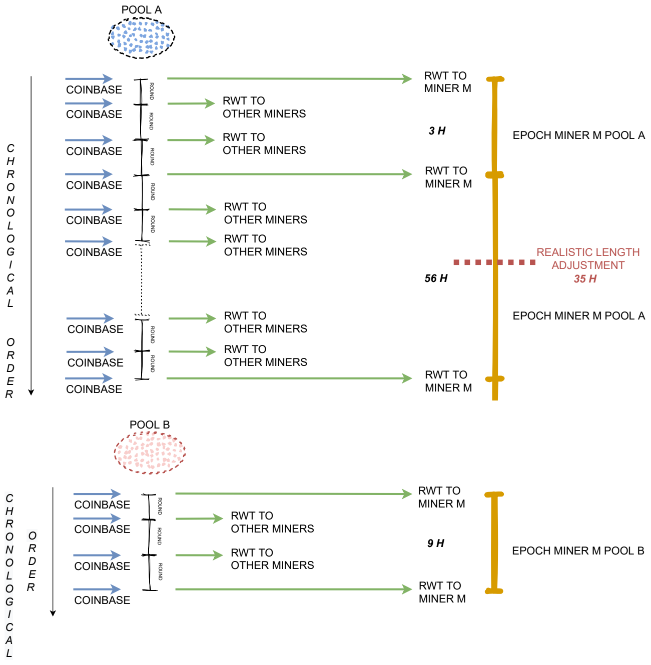


Figure 25: Hopping dynamic in a minimum rewarding threshold environment, with length adjustment.

miner M			
	Date	Time	Pool
TX1	2020/05/01	01:00:00	Pool A
TX2	2020/05/01	03:00:00	Pool A
TX3	2020/05/02	23:00:00	Pool B
TX4	2020/05/03	12:00:00	Pool A
TX3	2020/05/03	08:00:00	Pool B

Table 23
Rewarding transactions received by miner *m*

LIMIT	35 h

miner M		
	Start	End
	Pool A	
E1	2020/04/29 - 14:00:00	2020/05/01 - 01:00:00
E2	2020/05/01 - 01:00:00	2020/05/01 - 03:00:00
E3	2020/05/02 - 01:00:00	2020/05/03 - 12:00:00
	Pool B	
E1	2020/05/01 - 12:00:00	2020/05/02 - 23:00:00
E2	2020/05/02 - 23:00:00	2020/05/03 - 08:00:00

Table 24
Working epochs of miner *m*

	Start	End	Pool
	miner L		
E1	2020/05/01 10:00:00	2020/05/01 16:00:00	Pool A
E2	2020/05/03 03:00:00	2020/05/03 20:00:00	Pool A
	miner M		
E1	2020/04/29 14:00:00	2020/05/01 01:00:00	Pool A
E2	2020/05/01 01:00:00	2020/05/01 03:00:00	Pool A
E3	2020/05/01 12:00:00	2020/05/02 23:00:00	Pool B
E4	2020/05/02 23:00:00	2020/05/03 08:00:00	Pool B
E5	2020/05/02 01:00:00	2020/05/03 12:00:00	Pool A
	miner N		
E1	2020/05/02 07:00:00	2020/05/02 22:00:00	Pool C
E2	2020/05/03 10:00:00	2020/05/04 01:00:00	Pool A
E3	2020/05/05 20:00:00	2020/05/06 09:00:00	Pool C
E4	2020/05/06 17:00:00	2020/05/07 01:00:00	Pool B
E5	2020/05/07 14:00:00	2020/05/07 23:00:00	Pool C

Table 25
Working epochs of miner L, M, N

lighted in red. The same reasoning is applied for epoch E1. It ends with reward TX1, but we do not know how long the miner worked, so we assume the maximum duration of a realistic epoch. This is not the case for the interval between TX1 and TX2, which can be considered a realistic epoch without modification, since the miner worked continuously during the interval. Having concluded the transactions from Pool A, we proceed to analyze those from Pool B to the same miner. There are only two transactions, TX3 and TX5, so there are two intervals of work for the miner in that pool, one ending with TX3 and one with TX5. For the first interval, we don't know when it started so the maximum time of a realistic epoch applies. For the second, it is below the limit, so epoch E2 corresponds precisely to the reward interval.

D. Hopping Detection Example

Hereafter is an example of the functioning of the first phase of our detection. Algorithms 6, 7 and 8 is considered.

Assume that Table 25 is showing the schedule of epochs for miner L, M and N.

For miner L, algorithm 6 would stop at *Condition* 6, since is unique the pool in which it worked. For miner M,

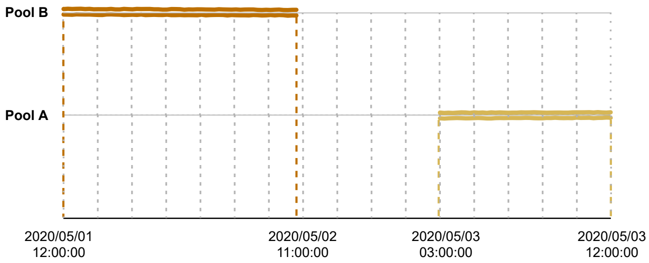


Figure 26: Overlapped epoch of a cross-epoch hopper that performed hopping between two pools.

instead, *Condition* 6 is verified, so it proceed combining all epochs with the previous ones:

- E2 with E1: *Condition* 7 not verified as they are in the same pool
- E3 with E1: *Condition* 7 verified, but *Condition* 8 is not, as E1 ends before E3
- E3 with E2: *Condition* 7 verified, but *Condition* 8 is not, as E2 ends before E3
- E4 with E1: *Condition* 7 verified, but *Condition* 8 is not, as E1 ends before E4
- E4 with E2: *Condition* 7 verified, but *Condition* 8 is not, as E2 ends before E4
- E4 with E3: *Condition* 7 verified, but *Condition* 8 is not, as E3 ends before E4
- E5 with E1: *Condition* 7 not verified, same pool
- E5 with E2: *Condition* 7 not verified, same pool
- E5 with E3: both *Condition* 7 and *Condition* 8 verified, as E5 start before E3 ends
- E5 with E4: both *Condition* 7 and *Condition* 8 verified, as E5 start before E4 ends

Lastly, *Condition* 9 is not verified, as miner M presents some epoch overlapped. It is an cross-epoch hopper and therefore it is not added to the set of intra-epoch hoppers. The Figure 26 shows the placement and overlap of its work epochs in the different pools. On the opposite, for miner N, *Condition* 8 is never verified as all epochs begin after all the previous are finished. Therefore, at *Condition* 9, miner N doesn't present overlapped epochs, even though it presents epochs in different pools. It is considered an intra-epoch hopper and added to $intra - epoch\ hoppers\ I$.

For miner N, algorithm 7 investigates the jumping behaviours and counts how many time a pool is been left. In particular:

- from E1 to E2: the *Increment rule* is applied, as from Pool C the miner goes to Pool A
- from E2 to E3: the *Increment rule* is applied, as from Pool A the miner goes to Pool C

Intra-epoch hopper N	
	Switches
Pool A	1
Pool B	1
Pool C	3

Table 26
Jumps of intra-epoch hopper N.

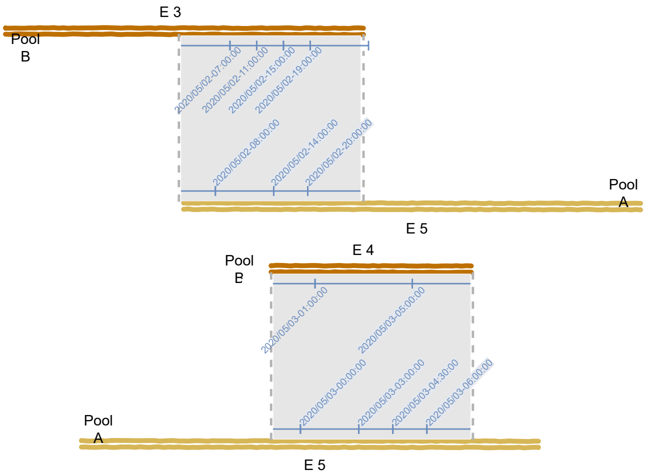


Figure 27: Windowing between miner M epochs.

- from E3 to E4: the *Increment rule* is applied, as from Pool C the miner goes to Pool B
- from E4 to E5: the *Increment rule* is applied, as from Pool B the miner goes to Pool C

Miner N is associated with three pools and with a jumps migration as displayed in Table 26. Given the time period in the example, the behavior of miner N can be considered strategic, since it changes pools back and forth quite a few times.

Given the overlaps presented by miner M, algorithm 8 analyzes the overlapping period between each couple of epochs and build the windows accordingly. looking at Figure 27, one can see that epoch E3 and E5 satisfy *Condition* 10, as E3 starts before E5, and *Condition* 11, as E3 also ends before E5. Instead, E4 and E5 satisfy *Condition* 12, as E4 starts after E5, but not *Condition* 13, as E4 ends before E5.

Also the respective coverage percentages are calculates, as shown in Table 27 and the rounds carried out by the pools in those intervals are stored individually to each epoch.

References

[1] Ametrano, F., 2014. Hayek money: The cryptocurrency price stability solution. SSRN Electronic Journal doi:10.2139/ssrn.2425270.

[2] Belotti, M., Božić, N., Pujolle, G., Secci, S., 2019. A vademecum on blockchain technologies: When, which, and how. IEEE Communications Surveys Tutorials 21, 3796–3838. doi:10.1109/COMST.2019.2928178.

[3] Belotti, M., Kirati, S., Secci, S., 2018. Bitcoin pool-hopping detection, in: 2018 IEEE 4th International Forum on Research and Technology for Society and Industry (RTSI), pp. 1–6. doi:10.1109/RTSI.2018.8548376.

Cross-epoch hopper M			
	Pool	Coverage	Rounds
Window E3-E5	Pool B	63%	2020/05/02 07:00:00 2020/05/02 11:00:00 2020/05/02-15:00:00 2020/05/02-19:00:00
Window E3-E5	Pool A	63%	2020/05/02 08:00:00 2020/05/02 14:00:00 2020/05/02 20:00:00
Window E4-E5	Pool B	100%	2020/05/03 01:00:00 2020/05/03 05:00:00
Window E4-E5	Pool A	23%	2020/05/03 00:00:00 2020/05/03 03:00:00 2020/05/03 04:30:00 2020/05/03-06:00:00

Table 27
Windows resulting form working intervals overlaps.

[4] Canellis, D., 2019. Bitcoin has nearly 100,000 nodes, but over 50% run vulnerable code. URL: https://thenextweb.com/hardfork/2019/05/06/bitcoin-100000-nodes-vulnerable-cryptocurrency/.

[5] Chatzigiannis, P., Baldimtsi, F., Griva, I., Li, J., 2019. Diversification across mining pools: Optimal mining strategies under pow. CoRR abs/1905.04624. URL: http://arxiv.org/abs/1905.04624, arXiv:1905.04624.

[6] Cortesi, E., 2021. A new approach for bitcoin pool-hopping detection (code). URL: https://github.com/cedric-cnam/poolhopping.

[7] Gao, S., Li, Z., Peng, Z., Xiao, B., 2019. Power adjusting and bribery racing: Novel mining attacks in the bitcoin system, in: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, Association for Computing Machinery, New York, NY, USA. p. 833–850. URL: https://doi.org/10.1145/3319535.3354203, doi:10.1145/3319535.3354203.

[8] JCTheMiner, 2020. PROP vs. PPLNS vs. PPS Mining Pool Reward Systems. URL: https://mintpond.com/b/prop-vs-pplns-vs-pps-mining-pool-reward-systems.

[9] Jiao, Z., Tian, R., Shang, D., Ding, H., 2018. Bicomp: A bilayer scalable nakamoto consensus protocol. CoRR abs/1809.01593. URL: http://arxiv.org/abs/1809.01593, arXiv:1809.01593.

[10] Kogias, E.K., Jovanovic, P., Gailly, N., Khoffi, I., Gasser, L., Ford, B., 2016. Enhancing bitcoin security and performance with strong consistency via collective signing, in: 25th USENIX Security Symposium (USENIX Security 16), USENIX Association, Austin, TX. pp. 279–296. URL: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/kogias.

[11] Mad, 2020. How many bitcoins are left to mine? URL: https://blog.bitnovo.com/en/how-many-bitcoins-are-left-to-mine/.

[12] Meiklejohn, S., Pomarole, M., Jordan, G., Levchenko, K., McCoy, D., Voelker, G.M., Savage, S., 2013. A fistful of bitcoins: Characterizing payments among men with no names, in: Internet Measurement Conference, Association for Computing Machinery, New York, NY, USA. p. 127–140. URL: https://doi.org/10.1145/2504730.2504747, doi:10.1145/2504730.2504747.

[13] Meni Rosenfeld, 2012. What is pool hopping? URL: https://bitcoin.stackexchange.com/questions/5072/what-is-pool-hopping.

[14] Merkle, R.C., 1979. Secrecy, Authentication, and Public Key Systems. Ph.D. thesis. Stanford, CA, USA. AAI8001972.

[15] Nakamoto, S., 2009. Bitcoin: A peer-to-peer electronic cash system. URL: http://www.bitcoin.org/bitcoin.pdf.

[16] Raikwar, M., Gligoroski, D., Kralevska, K., 2019. Sok of used cryptography in blockchain. IEEE Access 7, 148550–148575. doi:10.1109/ACCESS.2019.2946983.

[17] Reid, F., Harrigan, M., 2011. An analysis of anonymity in the bitcoin system, in: 2011 IEEE Third International Conference on Privacy, Security, Risk and Trust and 2011 IEEE Third International Conference on Social Computing, pp. 1318–1326. doi:10.1109/PASSAT/SocialCom.2011.79.

[18] Romiti, M., Judmayer, A., Zamyatin, A., Haslhofer, B., 2019. A deep dive into bitcoin mining pools: An empirical analysis of mining shares. CoRR abs/1905.05999. URL: http://arxiv.org/abs/1905.05999, arXiv:1905.05999.

[19] Rosenfeld, M., 2011. Analysis of bitcoin pooled mining reward systems. CoRR abs/1112.4980. URL: http://arxiv.org/abs/1112.4980, arXiv:1112.4980.

[20] Shi, H., Wang, S., Hu, Q., Cheng, X., Zhang, J., Yu, J., 2019. Hopping-proof and fee-free pooled mining in blockchain. CoRR abs/1912.11575. URL: http://arxiv.org/abs/1912.11575, arXiv:1912.11575.

[21] Shi, H., Wang, S., Hu, Q., Cheng, X., Zhang, J., Yu, J., 2021. Fee-free pooled mining for countering pool-hopping attack in blockchain. IEEE Transactions on Dependable and Secure Computing 18, 1580–1590. doi:10.1109/TDSC.2020.3021686.

[22] Tovanich, N., Soulié, N., Heulot, N., Isenberg, P., 2021a. An empirical analysis of pool hopping behavior in the Bitcoin blockchain, in: ICBC 2021 : IEEE International Conference on Blockchain and Cryptocurrency, IEEE Communications Society (ComSoc), Sydney (Virtual Conference), Australia. URL: https://hal.archives-ouvertes.fr/hal-03163006.

[23] Tovanich, N., Soulié, N., Isenberg, P., 2021b. Visual analytics of bitcoin mining pool evolution : on the road toward stability?, in: 3rd International Workshop on Blockchains and Smart Contracts (BSC 2020-2021), held in conjunction with the 11th IFIP International Conference on New Technologies, Mobility and Security (IFIP NTMS 2021), Paris, France. URL: https://hal.archives-ouvertes.fr/hal-02902465.

[24] Wang, W., Hoang, D.T., Xiong, Z., Niyato, D., Wang, P., Hu, P., Wen, Y., 2018. A survey on consensus mechanisms and mining management in blockchain networks. CoRR abs/1805.02707. URL: http://arxiv.org/abs/1805.02707, arXiv:1805.02707.

[25] Web-page, . Transactions — Bitcoin. URL: https://developer.bitcoin.org/devguide/transactions.html.

[26] Web-page, 2019a. Block hashing algorithm - Bitcoin Wiki. URL: https://en.bitcoin.it/wiki/Block_hashing_algorithm.

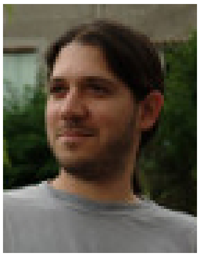
[27] Web-page, 2019b. Difficulty - Bitcoin Wiki. URL: https://en.bitcoin.it/wiki/Difficulty.

[28] Xia, J.z., Zhang, Y.h., Ye, H., Wang, Y., Jiang, G., Ying, Z., Xie, C., Kui, X.y., Liao, S.h., Wang, W.p., 2020. Supoolvisor: a visual analytics system for mining pool surveillance. Frontiers of Information Technology Electronic Engineering 21, 507–523. doi:10.1631/FITEE.1900532.

Eugenio Cortesi was born in Brescia, Italy in 1995. He obtained his computer engineering bachelor degree in 2018 and is about to get his Master degree in Computer Engineering in April 2021 with a specialization in ICT engineering, businness and innovation. He is currently working at Cnam on his Master thesis on Bitcoin analytics.

Francesco Bruschi is an assistant professor at the Politecnico di Milano, Italy. He holds a Ph.D. in Information Engineering from Politecnico di Milano, where he is also head of the Blockchain and Distributed Ledger Technology Observatory.

Stefano Secci is professor of networking at Cnam, Paris, France. He holds a Ph.D. degree from Tele-com ParisTech, and a M.Sc. and Ph.D. degrees in information and communications technologies from Politecnico di Milano, Italy. Website: http://cedric.cnam.fr/~seccis.



Sami Taktak is an associate professor at the Conservatoire National des Arts et Métiers (CNAM), Paris, France. He holds a M.Sc. degree in microelectronics and computer science and a Ph.D. degree in computer science from Sorbonne University, France.

Graphical Abstract

A New Approach For Bitcoin Pool-Hopping Detection

Eugenio Cortesi,Francesco Bruschi,Stefano Secci,Sami Taktak

Highlights

A New Approach For Bitcoin Pool-Hopping Detection

Eugenio Cortesi,Francesco Bruschi,Stefano Secci,Sami Taktak

- Complete algorithmic framework and related open-source code to detect pool-hopping miners in the Bitcoin network.
- Experimental evidence that, on the one hand, the pool-hoppers still obtain higher rewards in the long-run, even upon the introduction of new rewarding rules within mining pools that were meant to discourage pool-hopping.
- Experimental evidence that, on the other hand, the single rewarding transactions do not have higher amounts than static miners, differently than with old rewarding rules.