Measuring and Improving the Performance of the Bitcoin Network

MUHAMMAD ANAS IMTIAZ

Advisor: Prof. David Starobinski

Report prepared on: October 10, 2020

Abstract

Blockchain has emerged as a disrupting technology with applications in digital currencies, supply chain, health care, and the Internet of Things (IoT). Bitcoin is one of the most prominent and widely deployed blockchain with a market value of over $200 billion. Bitcoin is based on a decentralized architecture in which records, or *transactions*, are propagated to nodes in a network. These transactions eventually end up being included in *blocks* for the purpose of confirmation. The purpose of this thesis is to investigate the efficiency of the propagation mechanisms in Bitcoin through live measurements, and propose solutions that address inefficiencies.

As a peer-to-peer network by design, nodes in the Bitcoin network are free to independently leave and rejoin the network, creating an effect known as *churn*. When a node stays off the network, it does not receive information being propagated in the network. Our first research thrust is to investigate whether churn is present in the Bitcoin network, and how it affects the efficiency of the Bitcoin protocol. In our initial results, we find that the vast majority (97%) of Bitcoin nodes indeed exhibit churn. Through a novel statistical characterization of churn in the Bitcoin network, we find that the performance of the *compact block* protocol, a key component of the block propagation process, significantly degrades under churn. Chiefly, churning nodes experience a roughly five-fold increase in block propagation time.

In our second research thrust, we methodically research the effects of *orphan transactions* on the performance of the Bitcoin network. These are transactions for which parental income sources are missing. We first show that, surprisingly, orphan transactions tend to have fewer parents on average than non-orphan transactions. However, their missing parents have a lower fee, larger size, and lower transaction fee per byte than those of all other received transactions. Next, we show that when nodes join the Bitcoin network, as may occur under churn, a significant fraction of incoming transactions are orphan (about 25%). Last, we show that the current default buffer size for orphan transactions is too small and leads to unnecessary network overhead. This problem can be mitigated by increasing the default buffer size from 100 to 1000 transactions.

The above inefficiencies stem from the lack of consistency between the memory pools (*mempools*) of unspent transactions stored in each Bitcoin node. To address this issue, as part of our future work, we propose to develop efficient mechanisms for maintaining consistency of mempools across churning nodes, and integrate them with complementary approaches currently considered by the Bitcoin development community.

1 Introduction

Blockchain. A *blockchain* is an append-only list of records, or *blocks*, that are chained together with cryptographic hashes of previous blocks. This makes it almost impossible (as long as the underlying cryptography is secure) for an adversary to tamper with the data recorded in the blockchain without controlling a significant amount of resources. having to do significant work [4, 50, 59]. Blockchain is decentralized; it reduces the amount of trust placed in a single actor and requires an entire network of blockchain users to establish and maintain consensus, often with economic incentives.

Bitcoin. The Bitcoin cryptocurrency, originally introduced by Satoshi Nakamoto in 2008 [50] as a peer-to-peer electronic payment system, is currently used for buying and selling a wide variety of goods in different markets across the globe. At the time of writing, the cryptocurrency has a total market capitalization of well over $200 billion [2].

Bitcoin uses a blockchain to record all transactions that take place in the Bitcoin network [22]. Each new transaction is broadcast over the network, and thereafter recorded by every node in its local memory pool (known as a *mempool*) for subsequent consensus-based validation. By design, a new block containing transactions is created (by a *mining* node) and propagated over the network's nodes roughly once every ten minutes [66].

Churn and block propagation. A key challenge is reducing the propagation time of blocks. Indeed, the consequences of slower block propagation times include an increase of *forks*, wherein several blocks are mined independently and distributed before the network nodes accept one of the blocks as the head of the blockchain while the other blocks become orphan. This issue leads to periods of ambiguity, during which different nodes in the network have different views of the blockchain. An adversary may leverage such ambiguities for certain attacks, such as a double-spending attack [22]. Orphan blocks also lead to a waste of computational resources for nodes that have mined them and nodes that have mined on top of them.

To address this challenge, the *compact block relay* protocol [21], described in greater detail in subsection 2.1, has been proposed and is currently implemented on the standard Bitcoin Core reference implementation. This protocol aims to decrease the propagation time to the broader network by reducing the amount of data propagated between nodes.

Yet, like any peer-to-peer network, it is also important that the Bitcoin network be able to support a high rate of *churn* [63], that being the rate at which nodes independently enter and leave the network. In fact, Satoshi's white paper envisions that Bitcoin nodes "can leave and rejoin the network at will" [49]. In other words, the network should be able to quickly propagate blocks to all current nodes, even when some of these nodes frequently enter and leave the network. We look for a characterization and impact of churn in the Bitcoin network. Especially, does there exist churn in the Bitcoin network? If yes, then to what extent? Does it affect the propagation of block in the network? More precisely, does it increase the propagation times of blocks? Can we make the protocol resilient to churn? To the best of our knowledge, these questions remained largely unanswered prior to this work.

Orphan transactions. Before relaying a transaction to its peers, a node in the Bitcoin network must confirm that the transaction has verified currency input from its *parent* transactions. If parents of a transaction are not in the node's mempool or local blockchain, then the transaction is classified an *orphan*, and it is not relayed further until the parents arrive. We seek to more precisely understand the context under which a transaction becomes an orphan, including the properties of parent transactions that produce this effect. Specifically, to *what* extent orphan transactions are prevalent in the Bitcoin network? What are the factors that make a transaction orphan? What is the impact of an orphan transaction on the performance of the Bitcoin ecosystem? Does an orphan transaction incur latency or communication overhead? If so, can one reduce this overhead? There exists no work, to the best of our knowledge, that reasonably answers these questions.

Related work. Churn in different peer-to-peer networks has been widely studied, characterized and mod-

eled [31,32,42,62,63,67,68], though it has received little attention in relation to the Bitcoin network. While some previous works on Bitcoin do consider churn in their models [19,38,52], they do not seek a full characterization and evaluation of its impact. Undeniably, questions remain about the extent of churn in the Bitcoin network and its effect on block propagation.

Ozisik et al. [55] propose the *Graphene* protocol, which couples an Invertible Bloom Lookup Table (IBLT) [28] with a Bloom filter in order to send transactions in a package smaller than a compact block. According to the authors, the size of a Graphene block can be a fifth of the size of a compact block, and they provide simulations demonstrating their system. This block propagation concept has recently been merged into the Bitcoin Unlimited blockchain. However, similar to the compact block protocol, Graphene assumes a large degree of synchronization between mempools of sending and receiving peers. In case of missing transactions, the receiving peer requests larger IBLTs from the sending peer, thus potentially adding significant propagation delay.

Naumenko et al. [51] propose the *Erlay* transaction dissemination protocol, the aim of which is to reduce the consumption of bandwidth due to dissemination of transactions across nodes in the Bitcoin network. The protocol uses *set sketches* to perform set reconciliation across mempools of nodes. The authors do not evaluate the protocol in the presence of churn, and it is unclear whether Erlay would perform efficiently when a node misses a large number of transactions from a block that it receives. We show in this work that a block received by a churning node can miss as many as 2,722 transactions.

Bitcoin transactions have received a fair amount of attention in the literature. Subset of this work have focused on elements such as an analysis of the transaction graphs [24, 26, 29, 44, 48, 54, 57], security of transactions [18,33,43,45,58,65], studies on transaction confirmation times [39,40,61,70], and the like.

Understanding the properties and behavior of orphan transactions, however, is a largely unexplored field. The closest works utilize orphan transactions as a side-channel for topology inference [23], and for denial of service attacks on the Bitcoin network [5,46]. However, many of the performance questions regarding orphan transactions remain unanswered. For a detailed discussion of other related works, please refer to sections II-B in [35,36].

Contributions. In this Ph.D. research, we aim to investigate the issues raised above. We first present the key contributions obtained so far, and then discuss areas for future work.

Our first contribution is to systematically characterize churn in the Bitcoin network. Our characterization is based on measurements of the duration of time that nodes in the Bitcoin network are continuously reachable, i.e., *up session lengths*, and continuously unreachable, i.e., *down session lengths*. Our data show that out of more than 40,000 unique nodes on the network, over 97% leave and rejoin the network multiple times over a time span of about two months. In fact, the daily average churn rate in the Bitcoin network (i.e., the rate of oscillations between up and down sessions) exceeds 4 times per node. Our statistical analysis (cf. subsection 3.2) points out that, among several possible distributions, the *log-logistic distribution* and the *Weibull distribution* are the best fits for up session lengths and down session lengths, respectively.

Our second contribution involves an experimental evaluation of the behavior of the compact block protocol under realistic node churning behavior, leveraging our statistical characterization of churn to generate samples from the above distributions. We use our samples to emulate churn on nodes under our control in the *live Bitcoin network* (*i.e.,* on the Bitcoin `mainnet`), taking these nodes off the network and bringing them back on according to the sampled session lengths over a two week period. The performance of the system is measured by means of a novel framework that we develop for logging the internal behavior of a Bitcoin node and share for public use.

Our analysis, compared against a control group of nodes that are continuously connected to the network, shows that the performance of the compact block protocol significantly degrades in the presence of churn. Specifically, the churning nodes see a significantly larger fraction of incomplete blocks as the control nodes (an average of 33.12% vs. 7.15% unsuccessful compact blocks). This is due to an absence of about 78 transactions on average for the churning nodes, versus less than 1 transaction for the control nodes. The

end result is that, on average, churning nodes require over five times as much time to propagate a block than their continuously connected counterparts (*i.e.,* 566.89 ms vs. 109.31 ms). The largest propagation delay experienced by blocks received by churning nodes is more than twice the largest propagation delay experienced by any block received by the control nodes(*i.e.,* 105.54 s vs. 46.14 s). These results confirm that churn can have a significant impact on block propagation in Bitcoin.

Our third contribution is to characterize orphan transactions in the Bitcoin network and identify the environment that produces them, based on a data set of 4.20×10^6 *unique* transactions (8.71×10^4 of which are orphans) received over the measurement period. Our analysis shows that 45% of transactions that are orphan at some point end up being included in the blockchain during the measurement period. Out of them, in 68% of the cases, at least one missing parent appears in the same block as the orphan transaction. Counterintuitively, we find out that orphan transactions have fewer parents on average than non-orphan transactions (which presumably would resulting in a greater probability that one of the parents is missing). Indeed, orphan transactions have 1.18 parents on average versus 2.20 for non-orphans transactions. We conclude that the number of parents does not suitably distinguish between orphan and non-orphan transactions. We then consider other metrics (*i.e.,* transaction fee, transaction size, and transaction fee per byte) to discern the distinction between these two types of transaction. Our analysis shows that missing parents of orphan transactions have, on average, smaller fees (*i.e.,* 5.56×10^3 vs. 9.91×10^3 satoshis), larger size (*i.e.,* 5.29×10^2 vs. 4.80×10^2 bytes), and smaller fee per byte (*i.e.,* 6.25 vs. 21.73 satoshis per byte) than all other received transactions. As a result, transactions with a smaller fee per byte are more likely to go missing and render their descendent transactions orphans.

Our fourth contribution is to study the impact of network and performance overhead caused by orphan transactions. We thus collect data from live nodes in the Bitcoin network with various orphan pool sizes (including the default of 100). Our measurements show that orphan transactions incur a significant network overhead (*i.e.,* number of bytes received by their node) when the orphan pool size is smaller. In effect, the pool fills up and transactions in the orphan pool are rapidly evicted to make room for new orphan transactions. As such, an orphan transaction may be added to the orphan pool multiple times as it is announced by different peers. We show that by slightly increasing the orphan pool size to 1000 transactions, we can dramatically reduce this network overhead without a distinguishable effect on node performance (in terms of computation and memory).

Our fifth contribution is to study the behavior of orphan transactions in nodes that are either new or rejoin the network after a protracted disconnection. We emulate this property by periodically clearing the mempools of affected nodes. Our measurements show that immediately after a node joins the network with an empty mempool, over 25% of the transactions that it receives become orphan. However, as the node stays on the network for longer, the fraction of transactions that become orphan falls rapidly. Similarly, over measurement periods of 12 hours, we find that roughly 50% of all transactions that become orphan are received within the first two hours after the node joins the network.

Finally, we propose a mempool synchronization scheme, *`MempoolSync`*, as a proof-of-concept to improve Bitcoin performance. Our evaluation of the scheme shows that periodically synchronizing mempools of churning nodes with highly-connected nodes leads to fewer missing transactions and smaller propagation delays. We propose future works based on this evaluation. In particular, we propose improving the current implementation of `MempoolSync` to enable a node to perform synchronization with multiple peers. In addition, we propose an evaluation of the Graphene and Erlay protocols in the evaluation of churn which does not exist at the time of writing. Based on these evaluations, a smart synchronization protocol can be developed that can select the most suitable synchronization protocol based on the number of differences between two nodes in the Bitcoin network.

2 Background

2.1 Bitcoin Background

Blocks and the mempool. Bitcoin's primary record-keeping mechanism is the *block*. It is a data structure that contains metadata about the block's position in the blockchain together with a number of associated transactions (typically a couple thousands [13]). A block is generated roughly every ten minutes through the *mining* process, and, once generated, the block and its transactions become a part of the Bitcoin blockchain. There is a probability that different nodes will incorporate different blocks in their version of the blockchain (a process known as a *fork*). These differences are reconciled over time in a competitive process.

In the interim time between when a transaction is announced and when it is included in a block, transactions are stored locally in the *mempool*. The mempool is a constantly changing data set that stores all the unconfirmed transactions waiting to be included in future blocks. It typically contains anywhere between 10^4 to 10^5 transactions, depending on network activity. Currently the mempool experiences as low as 1 and as high as 17 insertions per second [16]; the arrival of a new block also instigates many deletions from the mempool, between 1,300 to 2,400 transactions [14] per block.

Block propagation. Block Propagation is the process of communicating a newly mined block to the network. It is the backbone of Bitcoin's ability to maintain consensus on the current balances of address (wallets). When a new block is discovered, each Bitcoin node advertises the block to all of its neighboring peers.

There are currently two main block protocols in Bitcoin: the original protocol developed for the first implementation of Bitcoin and the *Compact Block Relay* Protocol (BIP 152) [21]. The original protocol is adequate for block propagation but may require significant network resources, typically close to 1 MB per block [15]. Since neighboring peers in the Bitcoin networks can be geographically distant, this approach is susceptible to large delays [56].

The *compact block relay* was developed in an effort to reduce the total bandwidth required for block propagation. As the name implies, the compact block is able to communicate all the necessary data for a node to reconstruct and validate one standard block. The compact block contains the same metadata as the normal block, but instead of sending a full copy of each transaction, it sends only a hash of the transaction. Depending on the number of inputs and outputs, a transaction is between 500 and 800 bytes [12], whereas the hashes used for the compact block are only 6 bytes per transaction [21], a significant bandwidth saving that relies on the assumption that the receiver already has the relevant transactions in its mempool and just needs to know in which blocks they belong. This trade-off makes a compact block much smaller in size than the original block at the cost of potentially needing extra round-trip communications for transactions whose hashes the receiver does not recognize (using the `getblocktxn`/`blocktxn` messages [17]).

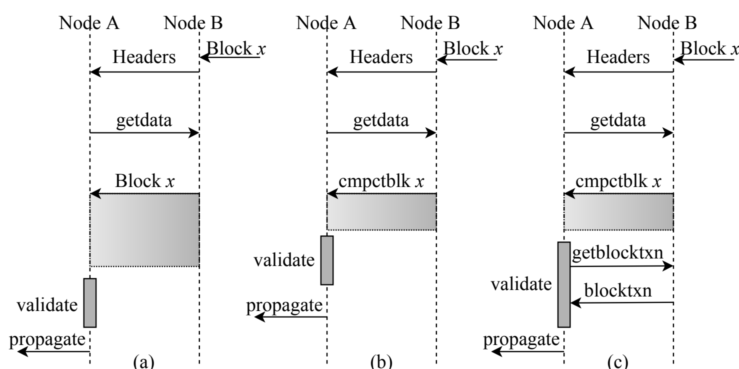

Fig. 1: Block propagation: (a) an *Original Satoshi block*, (b) a *Successful compact block*, (c) and an *Unsuccessful compact block*. In the last case, additional communications recover missing transactions from peers.

If a receiver's mempool contains all the transactions whose hashes are contained in a compact block that it received, then it will be able to successfully reconstruct the original block. However, if not all transactions

are already in the node's mempool then it will fail to reconstruct the block. When the compact block protocol fails, the extra round trips slow down block propagation and increases the risks of a fork in the blockchain. Fig. 1 illustrates the process.

Orphan transactions. A Bitcoin node may receive a transaction that spends income from one or more yet unseen parent transactions (*i.e.,* the parents are neither included in any of the previous blocks of the Bitcoin blockchain nor exist in the node's mempool). The node cannot accept the newly received transaction into its mempool until it can verify that the transaction spends valid Bitcoin, and it thus requests the missing parents from the peer that originally sent the transaction. In the meanwhile, the transaction is classified as an *orphan* transaction and added to an *orphan pool* that is maintained in the `mapOrphanTransactions` data structure in the Bitcoin core software. The transaction is not propagated forward to other peers until all of its missing parents are found.

Once the orphan transaction is added to the orphan pool, there are six cases that can cause its removal. These cases are listed below. For a detailed discussion and explanation of these cases, please refer to Appendix A.

1. Parent transactions received
2. Parent transactions in block
3. Orphan pool full
4. Orphan transaction timeout
5. Invalid orphan transaction
6. Peer disconnection

A transaction may get *stuck* [10] in mempools of nodes due to low transaction fees. That is, the transaction is not included in blocks and faces delays in confirmation. Bitcoin does allow the transactions to be modified to increase the fee [1], and the originator of the transaction may add a new input, *i.e.,* a new parent, as a spending source for the increased fee. The transaction may become orphaned if the new input is missing from the receiving node's mempool or local blockchain, and this transaction is then added to the orphan pool. We do not classify such orphan transactions separately because they do make it to the orphan pool.

3 Current work

```
Fri Nov  9 18:32:11 2018 1225883205020776 : CMPCTRECIVED - compact block received from xxx.xxx.xxx.xxx:8333
Fri Nov  9 18:32:11 2018 1225883205020776 : CMPCTBLKHASH -
↪   0000000000000000009636519364378c1db859d41f20f476bca69702360a1fe
Fri Nov  9 18:32:11 2018 1225883218437226 : CMPCTSAVED - /home/node/.bitcoin/expLogFiles/1806_cmpctblock.txt file
↪   created
Fri Nov  9 18:32:11 2018 1225883218469440 : DMPMEMPOOL --- Dumping mempool to file:
↪   /home/node/.bitcoin/expLogFiles/1806_mempoolFile.txt
Fri Nov  9 18:32:11 2018 1225883302280494 : FAILCMPCT - getblocktxn message sent for cmpctblock #1806
Fri Nov  9 18:32:11 2018 1225883302280494 : REQSENT - cmpctblock #1806 is missing 1238 tx
Fri Nov  9 18:32:11 2018 1225883302280494 : REQSAVED -  /home/node/.bitcoin/expLogFiles/1806_getblocktxn.txt file created
Fri Nov  9 18:32:11 2018 1225883459205249 : TXNFILLSUCCESS - blocktxn message recived and successfully filled missing
↪   TXN
```

Listing 1: An illustration of a log file produced by our data logging mechanism. The log file creates a time stamp (in nanoseconds) for certain events (e.g., receiving a compact block), and includes information such as the IP address of the sending peer and the number of transactions that a block is missing.

3.1 Data Collection Mechanism

To aid in understanding Bitcoin Core's behavior, we have developed a new log-to-file tool that produces human-friendly, easy-to-read text files. This logging system is open-source and we have made it available to the research community (`[34]/src/logFile.*`). This new logging mechanism allows one to isolate specific behaviors through select calls anywhere within the Bitcoin Core's source code.

The first step in development of this tool was to understand the structure of the Bitcoin Core software. We quickly realized that like many open-source software projects [6], the Bitcoin Core software lacked sufficient documentation. Therefore, a significant effort went into identifying *where* the events-of-interest were implemented in the Bitcoin Core software. By matching portions of software implementation with the "Bitcoin Protocol Documentation" [17], and tracing function calls within the software, we were able to pinpoint the implementation of these events-of-interest.

The logging system is designed such that it can record various events and the information associated with those events to a log file. For instance, when a compact block arrives, the system logs this event and saves the transaction hashes included in the compact block in a separate file with a unique identifier tying it to a log entry (as seen in Listing 1). The logging system is modular, and is, thus, fully compatible with any future development and enhancement.

The logging system allowed us to

1. Study efficiency of the compact block protocol in the presence of churn (cf. subsection 3.2), and

2. Characterize orphan transactions and analyze network overhead with varying orphan pool sizes (cf. subsection 3.3).

3.2 Churn in the Bitcoin Network

Nodes on the Bitcoin network may leave and rejoin the network independently resulting in a phenomenon known as *churn* [63]. As mentioned earlier, there did not exist a characterization of churn in the Bitcoin network prior to our work, *i.e.,* it was not known how long a node stays on or off the network once it joins or leaves the network, respectively. Therefore, we systematically characterized churn in the Bitcoin network [36]. This characterization was based on observation of node activity on the network.

Presence of churn. The site Bitnodes [69] continuously crawls the Bitcoin network and provides a list of all up nodes with an approximate 5 minute resolution as a JavaScript Object Notation (JSON). Each network snapshot contains the IP address, version of Bitcoin running, etc., of the nodes on the network that are up and reachable.

Our data, which was collected over a period of roughly 60 days, included a total of $14{,}674$ network snapshots. Our analysis of these snapshots showed that a total of $47{,}702$ *distinct* IP addresses appeared on the network during the 60-day period. If an IP address was found in two consecutive snapshots, we concluded that the IP address was *up*, and thus online for 10 minutes. Similarly, if the IP address was found in only one of the two consecutive network snapshots, we inferred that the node either left or rejoined the network. Finally, if IP addresses that were not found in any of the two consecutive network snapshots were designated as *down* (*i.e.,* offline) for 10 minutes. This allowed us to record the networked behavior of a node, *i.e.,* the duration of time it is on and off the Bitcoin network over the $14{,}674$ snapshots.

Out of $47{,}702$ distinct IP addresses observed on the network during the aforementioned time period, only $1{,}154$ (*i.e.,* 2.42% of the nodes) were online at all times. Nodes corresponding to the remaining IP addresses contributed to churn in the Bitcoin network. The average churn rate per node is 4.16 churns per day.

Statistical fitting of session lengths. Next, we performed a statistical distribution fitting of the up and down session lengths. Prior work $[20, 30, 41, 60]$ showed that session lengths in various peer-to-peer protocols

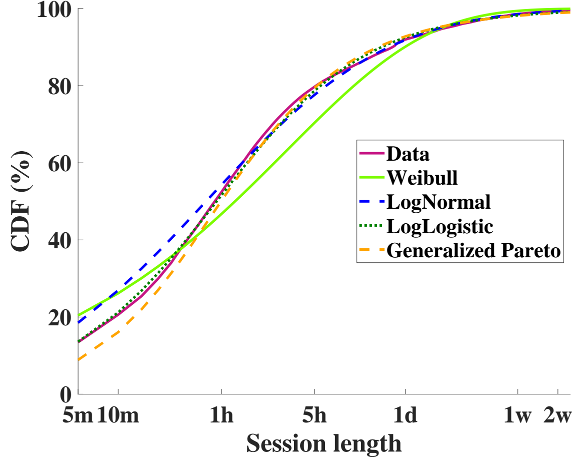

Fig. 2: Distribution fits for "up session" lengths.

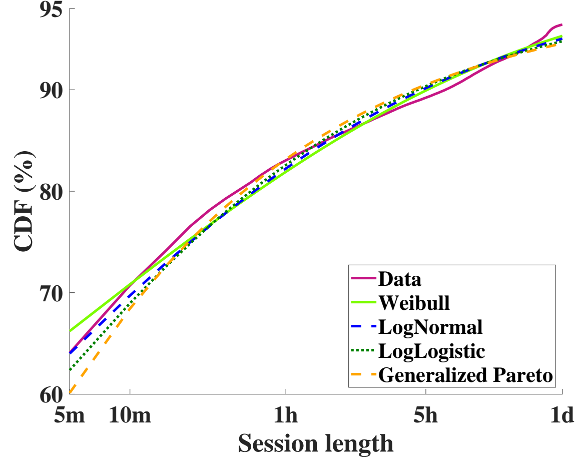

Fig. 3: Distribution fits for "down session" lengths.

exhibit a behavior similar to a heavy-tailed distribution. Therefore, in our statistical fitting, we focus on fitting heavy-tailed distributions to the data, specifically: the *generalized Pareto distribution*, *the log-normal distribution*, the *Weibull distribution* [27] and the *log-logistic distribution*. Nolan [53] shows that maximum likelihood estimation (MLE) of heavy-tailed distribution parameters is feasible. Hence, we use MATLAB's distribution fitting capabilities [37] to fit distributions based on the MLE criterion.

Judging from the combination of the R-squared values, the root mean squared errors, and visual inspection of distribution fits shown in Fig. 2 and Fig. 3, we concluded that the *log-logistic distribution*, given by

$$F_{(\alpha,\beta)}(x) = \frac{1}{1+(x/\alpha)^{-\beta}}, \tag{1}$$

where $\alpha > 0$ is the scale parameter, and $\beta > 0$ is the shape parameter, was the best fit for the up sessions. Similarly, the *Weibull distribution*, given by

$$F_{(\lambda,k)}(x) = \begin{cases} 1 - \mathrm{e}^{-(x/\lambda)^k} & x \geq 0 \\ 0 & x < 0, \end{cases} \tag{2}$$

where $\lambda > 0$ is the scale parameter, and $k > 0$ is the shape parameter, was the best fit for the down sessions.

Experimental analysis of compact block performance in the presence of churn. We evaluated the performance of block propagation, and especially the compact block protocol, in the presence of churning nodes, to realistically reflect the behavior of the Bitcoin P2P network.

• **Experimental setup.** We sampled up and down session lengths from the aforementioned distributions to emulate churn at our measurement nodes. In our experiments, that ran over a course of two weeks, four *churning* nodes (denoted by X_1, X_2, X_3, X_4) used sampled sessions lengths to emulate churn whereas four *control* nodes (denoted by C_1, C_2, C_3, C_4) remained connected to the network at all times. We used the log-to-file system (cf. subsection 3.1) to collect relevant data.

• **Statistics on missing transactions.** Churning nodes are generally missing far more transactions in blocks they are unable to reconstruct than the control nodes. We find that on average a churning node misses 78.08 transactions from a block with a standard deviation of 288.04 transactions, whereas a control node misses on average 0.87 transactions with a standard deviation of 10.78 transactions. Fig. 4 shows the CCDF

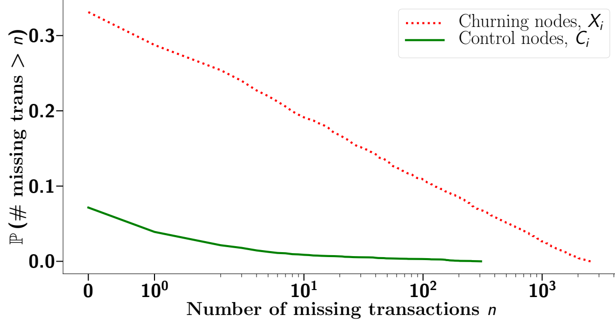

Fig. 4: CCDF of number of missing transactions in churning and control nodes.

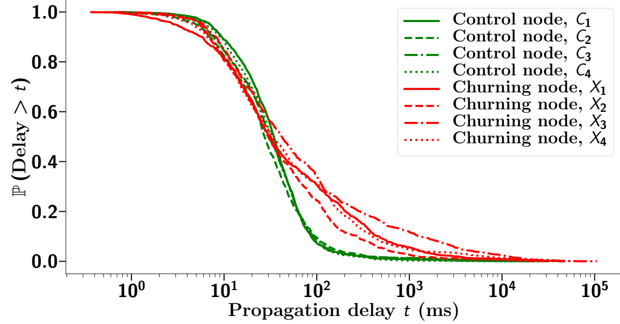

Fig. 5: Propagation delay across all blocks for both churning and control nodes.

of the number of missing transactions in blocks. From the figure, we observe that churning nodes may be missing up to thousands of transactions from a block they receive, while control nodes may miss at most a few hundred transactions.

• **Statistics on block propagation delay.** Next, we investigate whether and how transactions missing in a block delay the block's propagation. We compare the propagation delay of blocks received by churning nodes with the propagation delay of blocks received by control nodes. Blocks received by the control nodes experience an average propagation delay of 109.31 ms with a standard deviation of $1{,}066.15$ ms. Blocks received by the churning nodes, on the other hand, experience an average propagation delay of 566.89 ms with a standard deviation of $3{,}524.78$ ms. Fig. 5 shows the CCDF of propagation delays of blocks received by all nodes. From the figure, we observe that blocks received by control nodes rarely have large propagation delays.

Takeaways. We have shown through measurements that there does exist churn in the Bitcoin network. Our analysis shows that more than 97% of the nodes in the network churn at least once in a 60-day observation period. Next, we fit statistical distributions to the "up" and "down" session lengths of churning nodes. Using these distributions, we emulate churning nodes in the live Bitcoin network during our experiments. Data collected from these nodes show that in the presence of churn, nodes miss more transactions in blocks they received which creates larger block propagation delays. Therefore, it is evident that the compact block protocol on its own is not efficient in the presence of churn.

3.3 Orphan transactions in the Bitcoin Network

Characterization of orphan transactions. We performed the first ever characterization of orphan transactions in the Bitcoin network. Below we detail some of the main findings.

• **Measurement setup.** We connected live nodes to the Bitcoin network that record data relevant to orphan transactions using the log-to-file system described earlier. These nodes were configured with the default orphan pool size of 100 transactions.

• **Number of parents.** A natural conjecture is that a transaction with a large number of parents may be more likely to miss one or more parents than a transaction with, say, only a couple of parents. To this effect, we compare the number of parents of orphan transaction with the number of parents of all other non-orphan transactions.

Our analysis showed that, on average, an orphan transaction has 1.18 parents with a standard deviation of 4.78 transactions. On the other hand, a non-orphan transaction has, on average, 2.20 parents with a standard deviation of 11.84 transactions.

Surprisingly, orphan transactions do not necessarily have more parents than non-orphan transactions, and we are left to rely on other statistics, presented in the next few paragraphs, to characterize the orphan

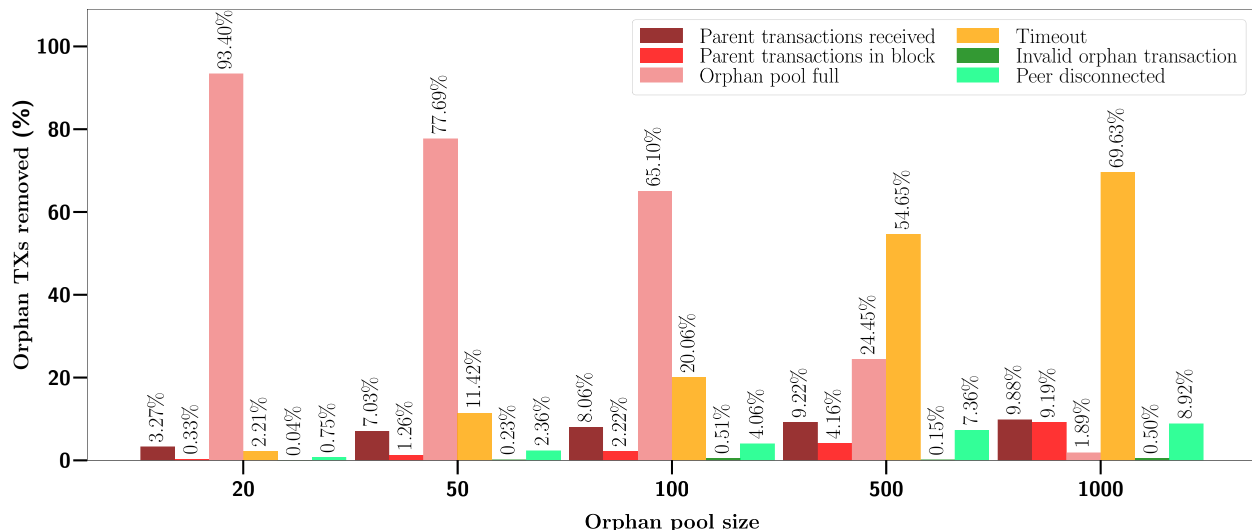

Fig. 6: Fraction of orphan transactions that are removed from the orphan pool due to each of the six causes across all nodes, under different pool sizes.

transactions.

• **Impact of transaction fees.** We compared the transaction fee of missing parents with all other transactions. We found that, on average, the transaction fee of a missing parent was 5.56×10^3 satoshis with a standard deviation of 7.17×10^4 satoshis. In comparison, the average transaction fee of all other transactions is 9.91×10^3 satoshis with a standard deviation of 5.53×10^4 satoshis. Therefore, a transaction is likely to become an orphan, if its missing parent has a transaction fee lower than that of other transactions.

• **Impact of transaction size.** We compared the transaction size of missing parents with all other transactions. Our analysis showed that, on average, missing parents have a size of 5.29×10^2 bytes with a standard deviation of 4.02×10^3 bytes. On the other hand, all other transactions have, on average, a size of 4.80×10^2 bytes with a standard deviation of 2.12×10^3 bytes. The statistics in this paragraph show that the missing parents of orphan transaction have, on average, a larger transaction size than all other transactions.

• **Orphan transactions in blocks.** Next, we examine what fraction of orphan transactions end up being included in blocks. Each node receives on average 1.58×10^3 blocks during the measurement period. Similarly, each node adds on average 4.64×10^4 *unique* transactions to its orphan pool. Out of these unique orphan transactions, on average, 2.06×10^4 transactions, *i.e.,* 44.50%, appear in blocks received by the nodes during the measurement period.

Further we find that only 11% of orphan transactions were recovered, *i.e.,* their missing parents were found, before they appear in blocks. A large fraction of the aggregated missing parents of orphan transactions appear in the same blocks as latter. Hence, many orphan transaction remain in that state until they are added into a block. This could lead to inefficiencies in the Bitcoin protocol [36] since transactions are not propagated to peers for as long as they remain orphan (cf. subsection 2.1).

Impact of orphan transactions. We next characterize the network and performance overhead incurred by orphan transactions, looking at both the default orphan pool size of 100 transactions, and various alternative pool sizes. In particular, we investigate network overhead under additions and removals of orphan transactions for different orphan pool sizes.

• **Removal of orphan transactions from orphan pool.** As specified in subsection 2.1, there are six different cases in which a transaction is removed from the orphan pool. Fig. 6 summarises the analysis of what fraction of transactions are removed from the orphan transaction falling within each of the six cases across the nodes with varying orphan pool sizes.

One trend is apparent: the major cases of transaction removal from the orphan pool are when the pool is full and when a transaction overstays its maximum allowed time in the pool. The figure clearly shows that as the size of the orphan pool increases, the major case of eviction of transactions from the orphan pool changes from the pool being full to the transactions timing out. That is, as the size of the orphan pool increases, more transactions are removed from the orphan pool due to timeout rather than a full orphan pool.

• **Addition of orphan transactions to orphan pool**. In the previous paragraphs, we saw that for smaller orphan pools, most transaction removals occur when the pool becomes full. However, this is not the case with orphan pools of larger sizes. Once an orphan transaction is removed from the orphan pool without being added to the mempool (cf. subsection 2.1), it *may* be added back to the orphan pool. This happens when, after its removal from the orphan pool, a peer announces the same transaction while its parents are still missing from the mempool or the blockchain.

We find that for smaller orphan pool sizes, identical transactions may be added several times to the orphan pool. This is likely because smaller orphan pool fill more quickly as the number of incoming orphan transactions grows. As such, transactions need to be removed more often from the orphan pool whilst they are still orphan. However, this transaction may be added to the orphan pool again if it is announced by a peer.

When the size of the orphan pool is larger than the default size of 100, the number of duplicate additions of transactions to the orphan pool goes down. This is likely due to the availability of space in the orphan pool for new orphan transactions; fewer transactions need to be evicted from the orphan pool.

• **Network overhead.** We next estimate the network overhead (*i.e.,* the number of bytes received) caused by receiving duplicate orphan transactions from peers. Our analysis shows that nodes with a smaller orphan pool size incur a larger network overhead due to the repeated addition of orphan transactions to the orphan pool. On the contrary, nodes with an orphan pool of larger size incur minimal network overhead, since the number of duplicate orphan transactions received is smaller.

Orphan transactions in nodes joining the network. A new node that joins the Bitcoin network has an empty mempool. Similarly, a node that stays off the Bitcoin network for a long period has a *stale* mempool, meaning that transactions in its mempool are not useful and are discarded when it rejoins the network. This is primarily because such transactions are already included in blocks that are created while the node is away from the network.

We observe that when a node starts up, a large fraction of transactions (*i.e.,* above 25%) are added to the orphan pool. However, as the node stays connected, the fraction of transactions that become orphan is reduced significantly.

Takeaways. We have performed the first ever characterization of orphan transactions in the Bitcoin network. Our results show that orphan transactions tend to have fewer parents than other transactions. Missing parents of orphan transactions are likely to have smaller transaction fees and larger sizes as compared to other transactions. We also report that a large fraction of orphan transactions appear in blocks where a significant portion of their missing parents are found in the same block. We have characterized the network overhead of orphan transactions with varying orphan pool sizes and shown that nodes with larger orphan pools experience lower network overhead. Finally, we have studied the behavior of nodes that are off the network for a long time or that have joined the network for the first time. Our findings show that such nodes experience as many as over 25% of all orphan transactions they receive as soon as they start up. This fraction decreases significantly as the node stays on the network for longer.

4 Future Work

The experimental results from Section 3.2 make it clear that missing transactions add significant delay to the propagation of blocks. This problem is especially acute for churning nodes since their mempools

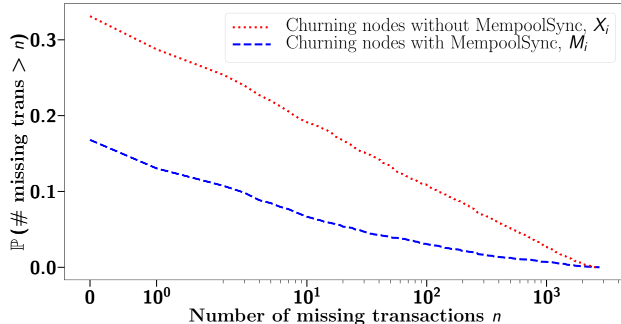

Fig. 7: CCDF of number of missing transactions across all blocks for all nodes.

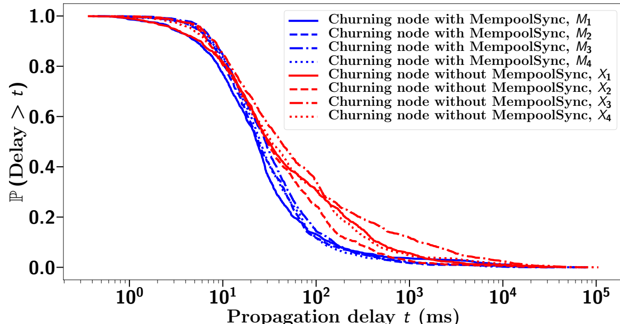

Fig. 8: CCDF of propagation delay across all blocks for all nodes.

often miss transactions. It is, therefore, evident that there should be a significant benefit in implementing a scheme that synchronizes mempools of Bitcoin nodes, thus keeping them up-to-date with transactions that they might have missed while being disconnected.

`MempoolSync`. As a proof-of-concept, we propose, implement and evaluate a new protocol, called *`MempoolSync`*, to keep the mempools of churning nodes synchronized with the rest of the network. The main goal of `MempoolSync` is to reduce the number of missing transactions in mempools and, consequently, the propagation delay of blocks. Note that `MempoolSync` does not attempt to minimize communication complexity, a well-known problem in the distributed computing literature [25, 28, 47] whose implementation we leave for future work. Rather, our implementation of `MempoolSync` into the Bitcoin Core demonstrates the key benefits of synchronizing the mempools of churning nodes with the rest of the network.

`MempoolSync` is designed to periodically synchronize the mempool of a churning node (*receiver*) with the mempool of a non-churning node (*sender*). The protocol leverages Bitcoin's existing functionality to package and send inventory (`inv`) messages, as well as request and propagate transactions. The sender selects transaction hashes from its mempool and packages them in a message (*inv*). The sender then sends the message to the receiver who, upon receiving the message, computes which of the hashes in the message are not present in its mempool. The receiver then requests the respective transactions from the sender (`getdata`), who in turn responds with the requested transactions (`tx`). The sender then waits for a configurable amount of time before repeating the process.

Experimental evaluation of `MempoolSync` in the presence of churn. We configured four nodes (denoted by M_1, M_2, M_3, M_4) to emulate churn with up and down sessions lengths independently sampled from the distributions obtained in Section 3.2 for each node. In addition, these nodes were also configured to accept `MempoolSync` messages.

• **Statistics on missing transactions.** We compare the number of transactions missing from compact blocks received by the churning nodes M_i and X_i. We find that churning nodes M_i that accept `MempoolSync` messages miss, on average, 21.30 transactions from blocks they received, with a standard deviation of 155.00 transactions. Churning nodes X_i that do not accept `MempoolSync` messages, on the other hand, miss, on average, 78.07 transactions from blocks they received, with a standard deviation of 288.04 transactions.

Fig. 7 shows the CCDF of the number of missing transactions in blocks. From the statistics and the figure, we conclude that to a large degree, `MempoolSync` successfully synchronizes the mempools of churning nodes. This synchronization results in far fewer missing transactions in compact blocks.

• **Statistics on block propagation delay.** With a smaller number of transactions missing from the compact blocks, one can expect that churning nodes implementing `MempoolSync` will have a smaller block propagation delay than churning nodes not implementing `MempoolSync`. Fig. 8 confirms this intuition.

Blocks received by churning nodes M_i experience, on average, a propagation delay of 249.06 ms with a standard deviation of $2{,}193.32$ ms. On the other hand, blocks received by churning nodes X_i experience, on average, a propagation delay of 566.89 ms with a propagation delay of $3{,}524.78$ ms.

Our experimental evaluation of `MempoolSync` confirm that implementing a scheme to synchronize mempools of churning nodes with highly-connected nodes reduces block propagation delays. We, therefore, propose the following future works along with a tentative time frame in which to complete the work.

1. **Improving `MempoolSync`.** Currently, `MempoolSync` works such that there is a 1-to-1 connection between a sender node and a receiver node. That is, a sender node is only connected to one receiver node, and vice versa. We plan to implement a feature in the Bitcoin software where it is possible for a node to 1) announce whether or not it provides or accepts the synchronization service; 2) be able to synchronize its mempool with multiple peers; and 3) select the best possible candidate to synchronize with out of all its peers (to reduce bandwidth overhead).

2. **Evaluation of Graphene and Erlay in the presence of churn.** The Graphene protocol, which is currently implemented in Bitcoin Unlimited [3], and Erlay, which is in review-phase for integration with Bitcoin software, have not been evaluated in the presence of churn. Therefore, we propose a set of experiments that independently perform this evaluation.

3. **Smart synchronization.** While `MempoolSync` has proven to be useful, it is still a one-way synchronization. That is, the sender does not know anything about the mempool of the receiver and may send useless information, thus wasting bandwidth. Erlay [51], CPISync [64], or IBLT [28] based synchronization schemes, such as Graphene [55], may be more efficient. However, none of these schemes have been evaluated in the presence of churn in the Bitcoin network. It is thus unclear whether any *one* of these schemes on its own is sufficient as a synchronization protocol on its own. For example, Erlay and CPISync perform well when the number of differences between the receiving node and the sender node is small. However, as the number of differences becomes large, which, according to our experimental results is quite possible, their computational complexity also grows to the extent where it may simply be sufficient to fall back to original Bitcoin slow-sync protocol. We, therefore, propose design and evaluation of a *smart* synchronization protocol that can select the most suitable synchronization protocol depending on the number of differences.

4. **Handling orphan transactions in synchronization scheme.** As described earlier, it is possible for orphan transactions to result in larger block propagation delays. `Package Relay` [8] is a proposed feature in Bitcoin that may take care of this issue [7]. However, the implementation of this proposal is a very long term plan. We propose to mimic similar (but not exactly the same) idea in any smart synchronization scheme that is designed.

References

[1] Bip-125. https://github.com/bitcoin/bips/blob/master/bip-0125.mediawiki. Online; Accessed: December 8, 2019.

[2] Bitcoin price, charts, market cap, and other metrics. https://coinmarketcap.com/currencies/bitcoin/. Online; Accessed: August 11, 2020.

[3] Bitcoin unlimited. https://www.bitcoinunlimited.info/. Online; Accessed: August 13, 2020.

[4] Bitcoin's kryptonite: The 51% attack. https://bitcointalk.org/index.php?topic=12435.0. Online; Accessed: August 24, 2020.

[5] Cve-2012-3789. https://en.bitcoin.it/wiki/CVE-2012-3789. Online; Accessed: February 12, 2020.

[6] How can open source projects be successful without documentation about their design or architecture? https://softwareengineering.stackexchange.com/questions/87994/how-can-open-source-projects-be-successful-without-documentation-about-their-des. Online; Accessed: August 8, 2020.

[7] Make orphan processing interruptible. https://bitcoincore.reviews/15644.html#l-75. Online; Accessed: August 10, 2020.

[8] Package relay. https://bitcoinops.org/en/topics/package-relay/. Online; Accessed: August 10, 2020.

[9] Sample Bitcoin configuration file. https://github.com/MrChrisJ/fullnode/blob/master/Setup_Guides/bitcoin.conf. Online; Accessed: November 11, 2019.

[10] Stuck bitcoin transaction. https://bitcointalk.org/index.php?topic=5135053.0. Online; Accessed: December 18, 2019.

[11] netprocessing.cpp. https://github.com/bitcoin/bitcoin/blob/0.18/src/net_processing.cpp. Online; Accessed: November 11, 2019.

[12] Analysis of bitcoin transaction size trends. https://tradeblock.com/blog/analysis-of-bitcoin-transaction-size-trends, 2015. Online; Accessed: November 17, 2018.

[13] Average number of transactions per block. https://blockchain.info/charts/n-transactions-per-block?timespan=2years, 2017. Online; Accessed: November 17, 2018.

[14] Average number of transactions per block. https://www.blockchain.com/en/charts/n-transactions-per-block?timespan=60days, 2017. Online; Accessed: November 17, 2018.

[15] Block size limit controversy.
https://en.bitcoin.it/wiki/Block_size_limit_controversy, 2017. Online; Accessed: November 17, 2018.

[16] Transaction rate. https://www.blockchain.com/charts/transactions-per-second?timespan=30days, 2017. Online; Accessed: November 17, 2018.

[17] Protocol documentation. https://en.bitcoin.it/wiki/Protocol_documentation, 2018. Online; Accessed: May 17, 2018.

[18] Elli Androulaki, Ghassan O Karame, Marc Roeschlin, Tobias Scherer, and Srdjan Capkun. Evaluating user privacy in Bitcoin. In *International Conference on Financial Cryptography and Data Security*, pages 34–51. Springer, 2013.

[19] Maria Apostolaki, Aviv Zohar, and Laurent Vanbever. Hijacking bitcoin: Routing attacks on cryptocurrencies. In *Security and Privacy (SP), 2017 IEEE Symposium on*, pages 375–392. IEEE, 2017.

[20] Fabian E Bustamante and Yi Qiao. Friendships that last: Peer lifespan and its role in p2p protocols. In *Web content caching and distribution*, pages 233–246. Springer, 2004.

[21] Matt Corallo. Compact block relay.
https://github.com/bitcoin/bips/blob/master/bip-0152.mediawiki, 2016.

[22] Christian Decker and Roger Wattenhofer. Information propagation in the bitcoin network. In *Peer-to-Peer Computing (P2P), 2013 IEEE Thirteenth International Conference on*, pages 1–10. IEEE, 2013.

[23] Sergi Delgado-Segura, Surya Bakshi, Cristina Pérez-Solà, James Litton, Andrew Pachulski, Andrew Miller, and Bobby Bhattacharjee. Txprobe: Discovering Bitcoin's network topology using orphan transactions. In *International Conference on Financial Cryptography and Data Security*, pages 550–566. Springer, 2019.

[24] Giuseppe Di Battista, Valentino Di Donato, Maurizio Patrignani, Maurizio Pizzonia, Vincenzo Roselli, and Roberto Tamassia. Bitconeview: visualization of flows in the Bitcoin transaction graph. In *2015 IEEE Symposium on Visualization for Cyber Security (VizSec)*, pages 1–8. IEEE, 2015.

[25] David Eppstein, Michael T Goodrich, Frank Uyeda, and George Varghese. What's the difference? efficient set reconciliation without prior context. *ACM SIGCOMM Computer Communication Review*, 41(4):218–229, 2011.

[26] Michael Fleder, Michael S Kester, and Sudeep Pillai. Bitcoin transaction graph analysis. *arXiv preprint arXiv:1502.01657*, 2015.

[27] Sergey Foss, Dmitry Korshunov, Stan Zachary, et al. *An introduction to heavy-tailed and subexponential distributions*, volume 6. Springer, 2011.

[28] Michael T Goodrich and Michael Mitzenmacher. Invertible bloom lookup tables. In *Communication, Control, and Computing (Allerton), 2011 49th Annual Allerton Conference on*, pages 792–799. IEEE, 2011.

[29] Alex Greaves and Benjamin Au. Using the Bitcoin transaction graph to predict the price of Bitcoin. *No Data*, 2015.

[30] Krishna P Gummadi, Richard J Dunn, Stefan Saroiu, Steven D Gribble, Henry M Levy, and John Zahorjan. Measurement, modeling, and analysis of a peer-to-peer file-sharing workload. *ACM SIGOPS Operating Systems Review*, 37(5):314–329, 2003.

[31] Xiaojun Hei, Chao Liang, Jian Liang, Yong Liu, and Keith W Ross. A measurement study of a large-scale p2p iptv system. *IEEE transactions on multimedia*, 9(8):1672–1687, 2007.

[32] Octavio Herrera and Taieb Znati. Modeling churn in p2p networks. In *Simulation Symposium, 2007. ANSS'07. 40th Annual*, pages 33–40. IEEE, 2007.

[33] Jordi Herrera-Joancomartí and Cristina Pérez-Solà. Privacy in Bitcoin transactions: new challenges from blockchain scalability solutions. In *International Conference on Modeling Decisions for Artificial Intelligence*, pages 26–44. Springer, 2016.

[34] Muhammad Anas Imtiaz. bitcoin-releases. `https://github.com/nislab/bitcoin-releases/tree/tnsm-churn`, 2020.

[35] Muhammad Anas Imtiaz, David Starobinski, and Ari Trachtenberg. Characterizing orphan transactions in the bitcoin network. In *2020 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*. IEEE, 2020.

[36] Muhammad Anas Imtiaz, David Starobinski, Ari Trachtenberg, and Nabeel Younis. Churn in the Bitcoin Network: Characterization and impact. In *2019 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, pages 431–439. IEEE, 2019.

[37] The MathWorks Inc. Fit probability distribution object to data - MATLAB fitdist. `https://www.mathworks.com/help/stats/fitdist.html`. Online; Accessed: November 06, 2018.

[38] Ghassan O Karame, Elli Androulaki, and Srdjan Capkun. Double-spending fast payments in bitcoin. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 906–917. ACM, 2012.

[39] Shoji Kasahara and Jun Kawahara. Effect of Bitcoin fee on transaction-confirmation process. *Journal of Industrial & Management Optimization*, 15(1):365–386, 2019.

[40] Yoshiaki Kawase and Shoji Kasahara. Transaction-confirmation time for Bitcoin: a queueing analytical approach to blockchain mechanism. In *International Conference on Queueing Theory and Network Applications*, pages 75–88. Springer, 2017.

[41] Jinyang Li, Jeremy Stribling, Robert Morris, and M Frans Kaashoek. Bandwidth-efficient management of dht routing tables. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*, pages 99–114. USENIX Association, 2005.

[42] Fuhong Lin, Changjia Chen, and Hongke Zhang. Characterizing churn in gnutella network in a new aspect. In *Young Computer Scientists, 2008. ICYCS 2008. The 9th International Conference for*, pages 305–309. IEEE, 2008.

[43] Yi Liu, Xingtong Liu, Chaojing Tang, Jian Wang, and Lei Zhang. Unlinkable coin mixing scheme for transaction privacy enhancement of Bitcoin. *IEEE Access*, 6:23261–23270, 2018.

[44] Dan McGinn, David Birch, David Akroyd, Miguel Molina-Solana, Yike Guo, and William J Knottenbelt. Visualizing dynamic Bitcoin transaction patterns. *Big data*, 4(2):109–119, 2016.

[45] Sarah Meiklejohn and Rebekah Mercer. Möbius: Trustless tumbling for transaction privacy. *Proceedings on Privacy Enhancing Technologies*, 2018(2):105–121, 2018.

[46] Andrew Miller and Rob Jansen. Shadow-bitcoin: Scalable simulation via direct execution of multi-threaded applications. In *8th Workshop on Cyber Security Experimentation and Test ({CSET} 15)*, 2015.

[47] Yaron Minsky, Ari Trachtenberg, and Richard Zippel. Set reconciliation with nearly optimal communication complexity. *IEEE Transactions on Information Theory*, 49(9):2213–2218, 2003.

[48] Malte Möser, Rainer Böhme, and Dominic Breuker. An inquiry into money laundering tools in the Bitcoin ecosystem. In *2013 APWG eCrime Researchers Summit*, pages 1–14. Ieee, 2013.

[49] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 03 2009.

[50] Satoshi Nakamoto et al. Bitcoin: A peer-to-peer electronic cash system. 2008.

[51] Gleb Naumenko, Gregory Maxwell, Pieter Wuille, Alexandra Fedorova, and Ivan Beschastnikh. Erlay: Efficient transaction relay for bitcoin. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 817–831, 2019.

[52] Till Neudecker, Philipp Andelfinger, and Hannes Hartenstein. A simulation model for analysis of attacks on the bitcoin peer-to-peer network. In *Integrated Network Management (IM), 2015 IFIP/IEEE International Symposium on*, pages 1327–1332. IEEE, 2015.

[53] John P Nolan. Maximum likelihood estimation and diagnostics for stable distributions. In *Lévy processes*, pages 379–400. Springer, 2001.

[54] Micha Ober, Stefan Katzenbeisser, and Kay Hamacher. Structure and anonymity of the Bitcoin transaction graph. *Future internet*, 5(2):237–250, 2013.

[55] A Pinar Ozisik, Gavin Andresen, George Bissias, Amir Houmansadr, and Brian Levine. Graphene: A new protocol for block propagation using set reconciliation. In *Data Privacy Management, Cryptocurrencies and Blockchain Technology*, pages 420–428. Springer, 2017.

[56] Giuseppe Pappalardo, Tiziana Di Matteo, Guido Caldarelli, and Tomaso Aste. Blockchain inefficiency in the bitcoin peers network. *arXiv preprint arXiv:1704.01414*, 2017.

[57] Dorit Ron and Adi Shamir. Quantitative analysis of the full Bitcoin transaction graph. In *International Conference on Financial Cryptography and Data Security*, pages 6–24. Springer, 2013.

[58] Tim Ruffing and Pedro Moreno-Sanchez. Valueshuffle: Mixing confidential transactions for comprehensive transaction privacy in Bitcoin. In *International Conference on Financial Cryptography and Data Security*, pages 133–154. Springer, 2017.

[59] John Salmon and Gordon Myers. Blockchain and associated legal issues for emerging markets. 2019.

[60] Subhabrata Sen and Jia Wang. Analyzing peer-to-peer traffic across large networks. In *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurment*, pages 137–150. ACM, 2002.

[61] Yonatan Sompolinsky and Aviv Zohar. Accelerating Bitcoin's transaction processing. fast money grows on trees, not chains. *IACR Cryptology ePrint Archive*, 2013(881), 2013.

[62] Daniel Stutzbach and Reza Rejaie. Towards a better understanding of churn in peer-to-peer networks. *Univ. of Oregon, Tech. Rep*, 2004.

[63] Daniel Stutzbach and Reza Rejaie. Understanding churn in peer-to-peer networks. In *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, pages 189–202. ACM, 2006.

[64] Ari Trachtenberg, David Starobinski, and Sachin Agarwal. Fast pda synchronization using characteristic polynomial interpolation. In *IEEE INFOCOM*, volume 3, pages 1510–1519. INSTITUTE OF ELECTRICAL ENGINEERS INC (IEEE), 2002.

[65] Qin Wang, Bo Qin, Jiankun Hu, and Fu Xiao. Preserving transaction privacy in Bitcoin. *Future Generation Computer Systems*, 2017.

[66] Bitcoin Wiki. Block. `https://en.bitcoin.it/wiki/Block`, 2016. Online; Accessed: November 17, 2018.

[67] Dong Yang, Yu-xiang Zhang, Hong-ke Zhang, Tin-Yu Wu, and Han-Chieh Chao. Multi-factors oriented study of p2p churn. *International Journal of Communication Systems*, 22(9):1089–1103, 2009.

[68] Zhongmei Yao, Derek Leonard, Xiaoming Wang, and Dmitri Loguinov. Modeling heterogeneous user churn and local resilience of unstructured p2p networks. In *icnp*, pages 32–41. IEEE, 2006.

[69] Addy Yeow. Bitnodes. `https://bitnodes.earn.com`. Online; Accessed: November 15, 2018.

[70] Yan Zhu, Ruiqi Guo, Guohua Gan, and Wei-Tek Tsai. Interactive incontestable signature for transactions confirmation in Bitcoin blockchain. In *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*, volume 1, pages 443–448. IEEE, 2016.

Appendix A

Once the orphan transaction is added to the orphan pool, there are six cases that can cause its removal (corresponding to lines 76, 2331, 2326−2330, 1609−1620, 876−906, 800−806, 40, 784−794, 627, 757−771, 1624−1632, and 1608 in the core implementation of `netprocessing.cpp` [11]):

1. **Parent transactions received.** The node receives a parent it requested from its peer. It then processes any orphan transactions that depend on the newly received transaction. All transactions that are no longer orphan are removed from the orphan pool and added to the mempool.

2. **Parent transactions in block.** The node receives a new block but does not directly check if it contains missing parents of an orphan transaction. Instead, for every transaction in the block, it checks whether an existing orphan transaction spends from an input of the former and removes the latter from the orphan pool if it does. This may be useful when orphan transactions and their missing parents are in the same block, or when a missing parent is received in a previous block.

3. **Orphan pool full.** By default, the size of the orphan pool is capped to a maximum of 100 orphan transactions. When the orphan pool is full, an orphan transaction is chosen at random and removed from the pool, and this transaction is not added to the mempool. The maximum size of the orphan pool can be modified at startup by using the `-maxorphantx` argument when running `bitcoind` or `bitcoin-qt`, or set in the `bitcoin.conf` configuration file [9].

4. **Timeout.** By default, an orphan transaction *expires* and is removed after 20 contiguous minutes in the orphan pool.

5. **Invalid orphan transaction.** The node deems that an orphan transaction is invalid when the missing parents of the orphan transaction have been received, but the orphan transaction itself may be non-standard or not have sufficient fee. Thus, this orphan transaction is not accepted to the mempool. Furthermore, not only the orphan transaction is removed from the orphan pool, but also the peer that originally sent the orphan transaction is punished, *i.e.,* no further transactions are accepted to the mempool from the peer in the current round.

6. **Peer disconnected.** When a peer disconnects from a node, all orphan transactions sent by this peer are removed from the orphan pool in the finalization step. This is likely because the node no longer expects to receive the parents it requested from the peer. The orphan transaction is not added to the mempool.