Investigating Orphan Transactions in the Bitcoin Network

Muhammad Anas Imtiaz, David Starobinski, and Ari Trachtenberg

***Abstract*—Orphan transactions are those whose parental income sources are missing at the time that they are processed. These transactions typically languish in a local buffer until they are evicted or all their parents are discovered, at which point they may be propagated further. To date, there has been little work in the literature on characterizing the nature and impact of such orphans, and yet it is intuitive that they should affect the performance of the Bitcoin network. This work thus seeks to methodically research such effects through a measurement campaign on live Bitcoin nodes. Our data show that about 45% of orphan transactions end up being included in the blockchain. Surprisingly, orphan transactions tend to have fewer parents on average than non-orphan transactions, and their missing parents have a lower fee, larger size, and lower transaction fee per byte than all other received transactions. Moreover, the network overhead incurred by these orphan transactions can be significant, exceeding 17% when using the default orphan memory pool size (*i.e.,* 100 transactions), although this overhead can be made negligible, without significant computational or memory demands, if the pool size is simply increased to 1000 transactions. Finally, we show that when a node with an empty mempool first joins the network, 25% of the transactions that it receives become orphan, whereas in steady-state this quantity drops to about 1%.**

***Index Terms*—Bitcoin, orphan transactions, characterization, transient behavior.**

I. Introduction

With a market cap of over 135 billion US dollars [2], the Bitcoin cryptocurrency has come a long way since its introduction as a peer-to-peer, electronic cash system by Satoshi Nakamoto in 2008 [3]. Nodes within the Bitcoin network exchange *transactions* to record purchases and sales using Bitcoin currency, one unit of which is further subdivided into 100 million *satoshis*. After such a transaction is created, it is propagated through the Bitcoin network, whose nodes add it to their local memory buffer called a *mempool*. Transactions stay in the mempool until confirmed by a Bitcoin miner [4] and added to a block in the common ledger known as a *blockchain*. Every day, hundreds of thousands of transactions are created and confirmed in the Bitcoin network [5], resulting in a total of over 480 million transactions since its inception [6].

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 a transaction's parents 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.

An earlier and shorter version of this paper appeared in the proceedings of the IEEE ICBC 2020 conference [1]. This research was supported in part by NSF under grant CCF-1563753.

The authors are with the ECE Department, Boston University, Boston, MA, 02215, USA. (email: {maimtiaz, staro, trachten}@bu.edu)

A. History

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 [7]–[13], security of transactions [14]–[19], studies on transaction confirmation times [20]–[23], and the like.

Understanding the properties and behavior of orphan transactions, however, is a largely unexplored field. The closest works have been on utilizing orphan transactions as a side-channel for topology inference [24], and for denial of service attacks on the Bitcoin network [25], [26]. However, many of the performance questions regarding orphan transactions remain: 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.

B. Contributions

Our first contribution in this paper 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. We discover that the intuition that orphan transactions may have larger numbers of parents than non-orphans (presumably resulting in a greater probability that one of the parents is missing) is misleading. Indeed, orphan transactions generally have *fewer* parents than all other transactions received during our measurements, averaging 1.18 parents (orphans) versus 2.20 (non-orphans). 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 transactions. Our analysis shows that missing parents of orphan transactions have smaller fees and larger size than all other received transactions. More precisely, a missing parent of an orphan transaction has an average transaction fee of 5.56×10^3 satoshis, and an average

transaction size of 5.29×10^2 bytes. By comparison, all other transactions have an average transaction fee of 9.91×10^3 satoshis and transaction size of 4.80×10^2 bytes. Next, we find that, on an individual level, missing parents of orphan transactions pay a fee of 6.25 satoshis per byte versus 21.73 satoshis per byte for all 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 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.

Our second 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). We also examine the effect of changing the timeout after which orphan transactions are removed from the pool. We do not observe marked improvement upon either increasing or decreasing the default value of 20 minutes.

Our third 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.

C. Road map

The rest of this paper is organized as follows: In Section II, we present preliminary background and related work. In Section III, we characterize orphan transactions by studying the properties of their parents and investigate presence of orphan transactions in blocks. We show the impact of orphan transactions with varying orphan pool sizes and varying orphan transaction timeouts in Section IV. In Section V, we study the behavior of orphan transactions in new nodes and nodes that have rejoined the network after a considerably long downtime. We present a discussion of our work, including limitations, in Section VI. Section VII concludes the paper and discusses potential areas for future work.

II. Background and related work

In this section, we provide relevant background material on the orphan transactions followed by a discussion of related work.

A. 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 (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` [27]):

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 [28].
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.

A transaction may get *stuck* [29] 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 [30], 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.

B. Related work

To the best of our knowledge, there is very little work in the Bitcoin research literature regarding orphan transactions. Nevertheless, the few works that do consider them highlight the potential value of the area, and the need for further work.

Miller and Jansen [25] take advantage of the fact that in the older version of Bitcoin (*i.e.,* v0.9.2), the protocol did not keep track of the peer that sent an orphan transaction. They propose that an adversary can leverage this vulnerability to mount a denial of service attack by sending a large number of orphan transactions to the victim node. The latter would be stuck verifying the transaction signatures of orphan transactions for a long time. However, this threat model is outdated since, in the current version, the Bitcoin protocol does keep track of the sender of an orphan transaction. The work also does not present a characterization of the orphan transactions.

Delgado-Segura et. al. [24] present *TxProbe*, a technique that makes use of orphan transactions to deduce the topology of the Bitcoin network. In this approach, an adversary creates a pair of double-spending transactions, and propagates each to a different node. The nodes try to propagate the double-spending orphan transaction to one another, if there exists an edge between the two. However, each of the receiving node rejects the incoming transaction as an invalid double-spending transaction. The adversary then sends a transaction that spends from one of the double-spending transactions to the node that received the corresponding double-spending transaction. This latter node will propagate the new transaction to the second node, if there exists an edge between the two. However, the second node will add the new transaction to its orphan pool, since it already rejected its parent earlier. The adversary can then probe the second node for the orphan transaction to establish a side-channel: if the node responds with the orphan transaction, the adversary deduces that there exists an edge between the two nodes that received the pair of double-spending transactions. The authors then extend this basic approach to a larger Bitcoin graph. Though this work presents an interesting side-channel in the Bitcoin network, it also does not characterize orphan transactions.

Earlier version of Bitcoin software did not place a limit on the number of orphan transactions that a node can store. Thus, an adversary could launch a denial of service attack by sending a large number of orphan transactions to a victim node, causing memory exhaustion and system failure. Furthermore, the Bitcoin software did not contain validation checks for the size of an orphan transaction. Hence, an adversary could create an orphan transaction with an arbitrarily large size and cause memory exhaustion at the victim node [31]. Both of these vulnerabilities were responsibly reported and fixed [26], [32]. While our work proposes increasing the size of the orphan pool, the current validation checks should ensure that this change will not enable denial of service attacks.

A preliminary version of this work was presented at the *2020 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)* [1]. The main differences between the aforementioned prior work and this work are as follows:

1) We introduce an entirely new section, Section V, where we investigate the transient behavior of orphan transactions in nodes that have joined the network for the first time or after a long downtime.
2) We provide more detailed characterizations of orphan transactions and their missing parents, in the new subsections III-F, III-G and III-H. In particular, we investigate the appearance of orphan transactions and missing parents in blocks as well as delays in receiving missing parents from peers, and the impact of low transaction fees.
3) In new subsection IV-F, we evaluate the impact of changing the default timeout value upon which orphan transactions are discarded from the orphan pools. Our study shows that the default timeout of 20 minutes is adequate.

III. Characterization of orphan transactions

We next detail our approaches toward characterizing the orphan transactions in the Bitcoin network. We begin with a presentation of our set up for data collection. Since a transaction becomes orphan due to the absence of one or more parents, we next focus on determining the characteristics of these missing parents. In particular, we compare the number of parents of orphan transactions with number of parents of all non-orphan transactions. Thereafter, we consider the differences between the transaction fee, transaction size, and transaction fee per byte of the missing parents of orphan transactions versus all other transactions. Finally, we investigate the presence of orphan transactions in blocks and the delay in receiving missing parents from peers, and observe the effect of low transaction fees on the propagation of transactions.

A. Measurement setup

For subsections III-B, III-C, III-D and III-E, we run two live full nodes N_1 and N_2 as part of the Bitcoin network, with the aim of collecting data for characterizing orphan transactions. Both nodes execute Bitcoin Core v0.18 [33] on the Linux Ubuntu 18.04.2 LTS distribution, running on Dell

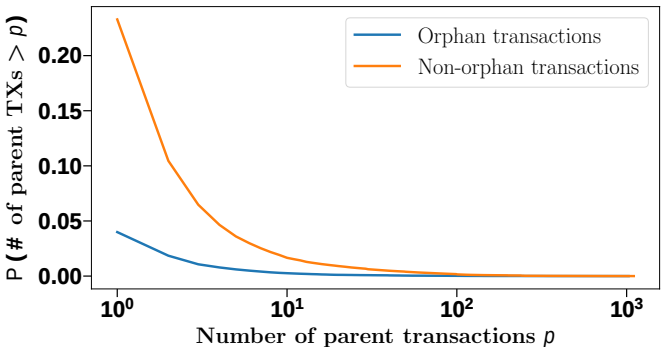

Fig. 1: Empirical complementary cumulative distribution function (CCDF) of (i) the number of parents of orphan transactions and (ii) number of parents of non-orphan transactions. In general, orphan transactions have fewer parents.

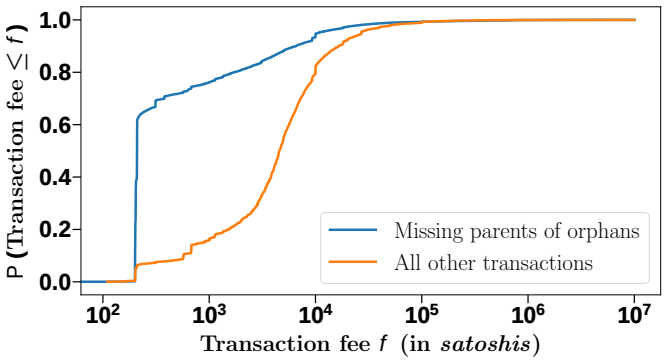

Fig. 2: Cumulative distribution functions (CDFs) of transaction fee of missing parents of orphan transactions, and transaction fee of all other transactions.

Inspiron 3670 desktops, each equipped with an 8$^{\text{th}}$ Generation Intel® Core i5−8400 processor (9 MB cache, up to 4.0 GHz), 1 TB HDD and 12 GB RAM. The nodes are connected to the Bitcoin network at all times with the default orphan pool size of 100. We collect relevant data, such as arrival of transactions, addition of transactions to the orphan pool, and the like, with the help of a log-to-file system [34], [35], for roughly 2 weeks over two rounds (November 18, 2019 11:00 AM to November 25, 2019 10:59 AM, and November 25, 2019 11:00 AM to December 02, 2019 10:59 AM).

In subsections III-F, III-G and III-H, we extend our measurement setup to four nodes with similar hardware and software specifications as mentioned above. The experiments run from July 6, 2020 3:00 PM for two weeks for subsections III-F and III-G, and from November 11, 2020 1:30 PM for roughly one week for subsection III-H.

B. Number of parents

Our first 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.

During the measurement period, the nodes receive an aggregate of 4.20×10^6 *unique* transactions with 9.23×10^6 parents. Of these, 8.71×10^4 are orphan transactions with 1.03×10^5 parents. These orphan transactions have an aggregate of 8.71×10^4 parents missing across the nodes. These nodes miss, on average, 1.23 parents per orphan transaction with a standard deviation of 4.68 parents. While only just above 2% of the received transactions become orphan, the total number is still significant.

Fig. 1 shows the complementary cumulative distribution functions (CCDF) of the number of parents of orphan transactions, and the CCDF of the number of parents of non-orphan transactions. We observe that our conjecture is flipped - the orphan transactions have a *smaller* number of parents. Indeed, only about 4% of orphan transactions have more than one parents, whereas roughly 25% of non-orphan transactions have more than one parent.

The most parents of an orphan transaction is 1.03×10^3, whereas this number is 1.10×10^3 for non-orphan transactions. 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 sections, to characterize the orphan transactions.

C. Transaction fee of missing parents

For each incoming transaction that is orphaned, we log the missing parent(s) that results in the transaction becoming orphan. We analyze and compare the transaction fees of these missing parents with all other transactions received by our nodes that are not a missing parent of an orphan transaction. We query the database maintained by the Bitcoin software for relevant data on transactions. Out of 8.71×10^4 missing parents, only about 3% are still missing by the end of the measurement period. Henceforth, we assume that this relatively small fraction does not pose a bias towards our findings.

Fig. 2 shows the cumulative distribution functions (CDFs) of transaction fees (in *satoshis*) of missing parents, and the CDF of transaction fees (in *satoshis*) of all other transactions received by the nodes. The figure shows that a majority of the missing parents have a lower transaction fee compared to all other transactions received. Indeed, 50% of missing parents have a transaction fee smaller than 210 satoshis. On the other hand, fewer than 6% of all other transactions have a transaction fee of smaller than 210 satoshis.

In fact, the average transaction fee of a missing parent is 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. Interestingly, 18 of the missing parents have *no* transaction fee at all (*i.e.,* 0 satoshis), whereas all other transactions received have a non-zero transaction fee.

Therefore, a transaction is likely to become an orphan, if its missing parent has a transaction fee lower than that of other transactions. As a future work, it would be interesting

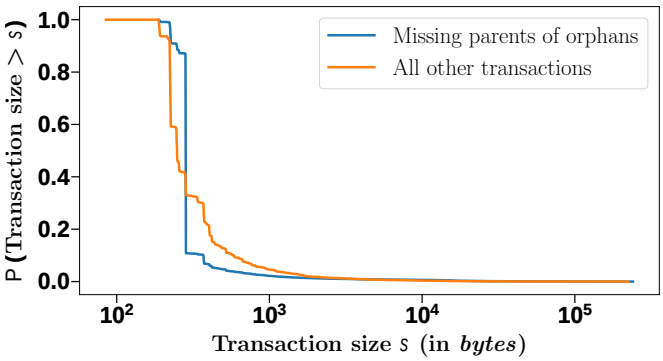

Fig. 3: CCDFs of transaction size of missing parents of orphan transactions, and transaction size of all other transactions.

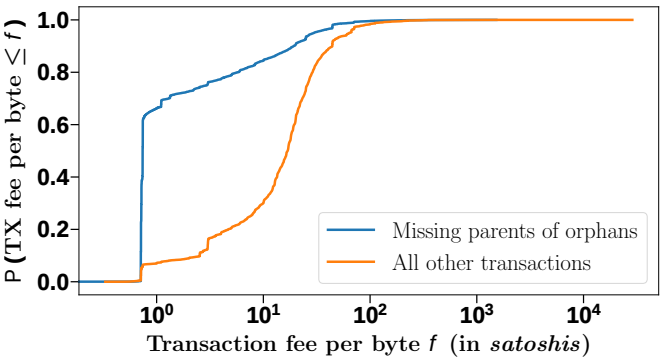

Fig. 4: CDFs of transaction fee per byte of missing parents of orphan transactions, and transaction fee per byte of all other transactions.

to deduce if there exists a threshold for the transaction fee below which all transactions become missing, *i.e.,* they are not relayed by the network.

D. Transaction size of missing parents

We next compare the sizes of missing parents of orphan transactions with the sizes of all other transactions. Do the missing parents of orphan transactions have a larger size than an average transaction?

Fig. 3 shows the CCDF of the size of missing parents of orphan transactions (in bytes) and the CCDF of the size of all other transactions. The figure shows that missing parents usually have a larger size than all other transactions. Roughly 90% of the missing parents have a size larger than 250 bytes, whereas only about 45% of all other transactions have a size larger than 250 bytes.

Missing parents of orphan transactions have a size between 1.88×10^2 and 2.40×10^5 bytes. By comparison, all other transactions have a size in the range of 8.50×10^1 to 2.24×10^5 bytes. In fact, 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 section show that the missing parents of orphan transaction have, on average, a larger transaction size than all other transactions. As in the previous section, we leave to future work the question whether there exists a size threshold above which transactions stop being propagated through the network.

E. Relating transaction fee to size of missing parents

We showed in subsections III-C and III-D that, in aggregate, missing parents tend to have a lower fee and a larger size than the average received transactions. However, it would be interesting to see if there exists a relation between the fee and size of each individual transaction.

To this end, Fig. 4 shows the CDF of transaction fee *per byte* (in satoshis) of missing parents and the CDF of transaction fee per byte of all transactions received. The figure shows that the missing parents generally have a lower transaction fee per byte when compared to all received transactions. Indeed, 80% of missing parents have a transaction fee per byte of 5.97 satoshis or less, whereas roughly 78% of all received transactions have a transaction fee per byte higher than 5.97 satoshis.

On average, missing parents have a transaction fee per byte of 6.25 satoshis with a standard deviation of 21.52 satoshis. On the other hand, all received transactions have a transaction fee per byte of 21.73 satoshis with a standard deviation of 47.13 satoshis.

Our data thus show that individual missing parents have a low transaction fee per byte. This could be because transactions with lower fees may not get properly propagated through the Bitcoin network [36], possibly because of configurable mempool size [37]. Note that nodes may choose not to accept transactions with a low transaction fee per byte to their mempool, and thereby not propagate them further [38].

F. 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 (we show in subsection IV-C that the same transaction may be added multiple times to the orphan pool - here we make sure to count such a transaction only once). 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. A block received during this period contains, on average, 2.17×10^3 transactions with a standard deviation of 7.31×10^2 transactions. Of these, an average of 13.06 transactions were orphan at some point during the measurement period, with a standard deviation of 25.90 transactions.

We next check whether orphan transactions were recovered (*i.e.,* removed from the orphan pool) before they appear in blocks. Specifically, we investigate whether missing parents of these orphan transactions were received from peers or not. Our analysis shows that only about 11% of the orphan transactions that appear in blocks were recovered before the respective block is received. For such orphan transactions, Fig. 5 shows, on an aggregate level, the CDF of the time elapsing from the

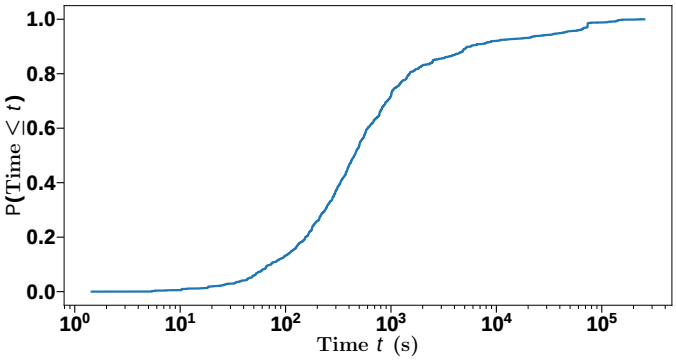

Fig. 5: CDF of time elapsing from the point a transaction is removed from the orphan pool because its missing parents are found till the block containing the said orphan transaction is received.

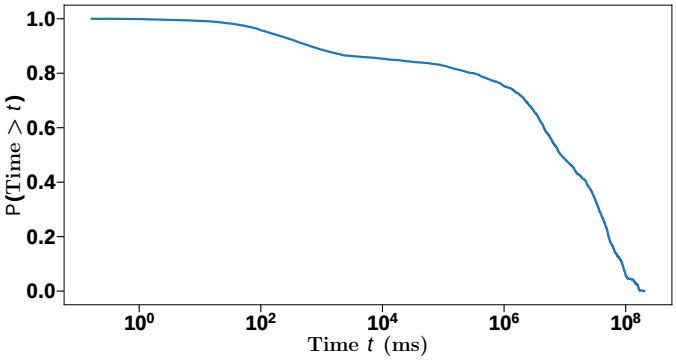

Fig. 6: CCDF of time elapsing from the point a transaction becomes orphan till one of its parents is found.

point a transaction is removed from the orphan pool because its missing parents are found till the block containing the said orphan transaction is received. On average, missing parents of such orphan transactions are found 5.70×10^3 seconds before the block containing the orphan transaction is received, with a standard deviation of 2.04×10^4 seconds.

We find that many missing parents (*i.e.,* on average, 67.58% of the total), of orphan transactions appear in the same block as the latter. The remaining missing parents (*i.e.,* 32.42% of the total) appear in a block received prior to the block containing the orphan transaction. Hence, many orphan transaction remain in that state until they are added into a block. This could lead to inefficiencies in the Bitcoin protocol [34], [35] since transactions are not propagated to peers for as long as they remain orphan (cf. subsection II-A).

G. Delay in receiving missing parents from peers

As noted in subsection II-A, when a node adds a transaction to its orphan pool, it sends requests for the missing parents to the peer that sent the transaction. Therefore, we next investigate, on an aggregate level, how long it takes for a requested missing parent of a transaction to be found once it is added to the orphan pool. In our experiment, orphan transactions have a total of 2.03×10^5 missing parents. Of these, only 3.24×10^4 are found during the measurement

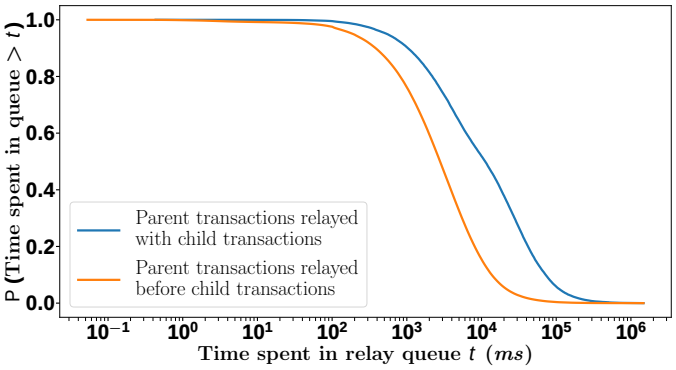

Fig. 7: CCDFs of time spent in relay queue of parent transactions that are relayed with and parent transactions that are relayed before their respective child transactions.

period (cf. subsection III-A) which represents about 15.98% of the total number of missing parents recorded.

Fig. 6 shows for the requested missing parents that are found (*i.e.,* sent by a peer), the CCDF of time elapsing from the point a transaction is added to the orphan pool till its missing parent is found (or one of the missing parents is found in the case of multiple parents). We observe that 10% of such missing parents are found within roughly 550 ms. Yet, on average, a missing parent is found within 2.89×10^7 ms, *i.e.,* roughly 8 hours after the respective child transaction was added to the orphan pool, with a standard deviation of 3.91×10^7 ms. We note that the average delay is high because many missing parents are found several hours after the respective child transaction becomes orphan. Indeed, about 50% of the missing parents are found at least 2 hours after the respective child transaction is added to the orphan pool. Similarly, roughly 35% of the missing parents have a delay larger than the mean.

H. Impact of transaction fee

The findings in subsection III-F showed that some parent transactions take a long time to be recovered. To explain this, we next perform an analysis of the propagation of transactions to show the effect of low transactions fees. For this purpose, we collect all transactions that are announced by our measurement nodes to their peers during the measurement period. Next, we identify pairs of transactions in our data set that have a child-parent relationship, *i.e.,* both the child and parent transactions were announced to the same peer. We compare and contrast two cases of interest, namely when a parent transaction is announced to a peer (i) *before* the child transaction; or (ii) *together* with its child transaction.

Recall that when a node receives a transaction, it performs various validation checks before adding the transaction to its mempool. Once the transaction passes all validation checks, it is added to the mempool as well as to a relay queue. The transaction is eventually retrieved from the relay queue and propagated to peers of the node. The transactions are retrieved from the queue in order of their transaction fees, *i.e.,* the transactions, grouped with their ancestors in the relay queue, with higher fees are announced first.

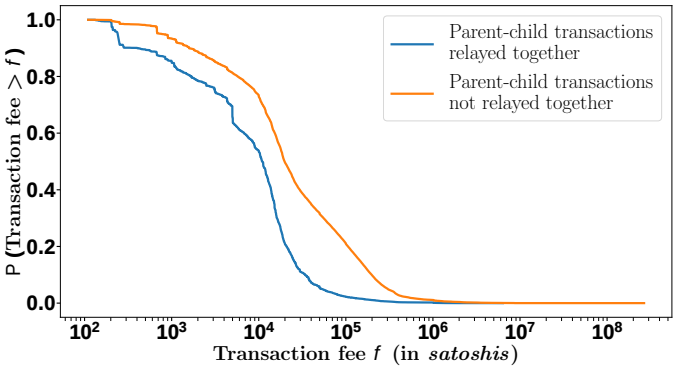

Fig. 8: CCDFs of transaction fee of parent transactions that are relayed with and parent transactions that are relayed before their respective child transactions.

Our first comparison is based on the delay from the point a transaction is added to the relay queue until it is sent out to peers of the node. We perform this analysis for both cases of interest identified earlier. Fig. 7 shows the CCDFs of time spent in the relay queue for both cases. The figure shows that parent transactions that are relayed together with their child transactions spend more time in the relay queue than parent transactions that are relayed before their child transactions. Indeed, the former spend, on average, 2.97×10^4 ms in the relay queue with a standard deviation of 6.14×10^4 ms. The latter, on the other hand, spend, on average, 6.42×10^3 ms in the relay queue with a standard deviation of 1.71×10^4 ms.

Our next comparison is depicted in Fig. 8, which shows the CCDFs of the transaction fee of the two types of parent transactions. The figure indicates that parent transactions that are announced to peers together with their child transactions usually have a smaller transaction fee than parent transactions that are announced to peers before their child transactions. We find that the former have, on average, a transaction fee of 2.02×10^4 satoshis with a standard deviation of 7.32×10^4 satoshis. The latter, on the other hand, have, on average, a transaction fee of 8.72×10^4 satoshis with a standard deviation of 3.32×10^5 satoshis.

The results of this section show that transactions with low transaction fees spend more time in the relay queue. Many such transactions are announced to peers only when their child transactions are also added to the queue. We hypothesize that the child and parent transactions together have enough fee to offset the low transaction fee of the parent transaction.

IV. Comparison of orphan transaction behavior with different orphan pool parameters

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. We begin with a presentation of our extended measurement setup, followed by an investigation of the network overhead under additions and removals of orphan transactions for different orphan pool sizes. Next, we discuss performance overhead that a larger orphan pool size may present. Finally, we present the effect of varying orphan transaction timeouts.

A. Measurement setup

For subsections IV-B, IV-C, IV-D and IV-E, we extend our measurement setup from subsection III-A to six live full nodes, running with identical hardware and software specifications as before. We run two rounds of experiments. In the first round, which runs from November 18, 2019 11:00 AM to November 25, 2019 10:59 AM, two nodes are configured with a default orphan pool size of 100 transactions (nodes N_1 and N_2), two nodes with an orphan pool size of 20 transactions (nodes N_3 and N_4), and the remaining two nodes with an orphan pool size of 50 transactions (nodes N_5 and N_6). In the second round, which runs from November 25, 2019 11:00 AM to December 02, 2019 10:59 AM, two nodes are configured with a default orphan pool size of 100 transactions (nodes N_1 and N_2), two nodes with an orphan pool size of 500 transactions (nodes N_3 and N_4), and the remaining two nodes with an orphan pool size of 1000 transactions (nodes N_5 and N_6). We have made all relevant logs generated during the experiments open source and accessible on GitHub [39].

Since our nodes are co-located, we want to verify that the nodes connect independently to outside peers in the network, and that our co-location does not impose a bias in the measurements. We achieve this by recording a node's connected peers over time, in one second intervals. We then check for common peers amongst the nodes throughout the measurement period, *i.e.,* both the first and the second rounds.

Fig. 9 and Fig. 10 show the common peers amongst nodes during the measurement period (*i.e.,* the first and second rounds of measurement respectively) as similarity matrices. A similarity score of 1.0 between two nodes indicates that both nodes have exactly the same peers; a similarity score of 0.0 indicates that the corresponding nodes have no common peers. The matrices in the figures qualitatively suggest that the six nodes have a very low number of peers in common, and therefore, do not present bias towards measurements.

In fact, the maximum number of peers that all six nodes have in common during the first round of measurements was 11 peers out of a maximum of 124 peers. On average, at any second during the measurement period, all six nodes have 8.30 peers in common with a standard deviation of 1.04 peers. Similarly, during the second round of measurements, the maximum number of peers that all six nodes have in common is 11 peers out of a maximum of 124 peers. On average, at any second during the measurement period, all six nodes have 8.51 peers in common with a standard deviation of 0.92 peers. These statistics confirm that nodes largely connect to, and interact with peers independently.

For subsection IV-F, we extend our measurement setup from subsection III-A to twelve live full nodes, running with identical hardware and software specifications as before. Our experiments run uninterrupted for two weeks from June 17, 2020 12:00 PM to July 1, 2020 11:59 AM. We configure two nodes with a timeout of 10 minutes, two nodes with a timeout of 15 minutes, four nodes with the default timeout of 20 minutes, two nodes with a timeout of 30 minutes, and the remaining two nodes with a timeout of 60 minutes. We configure all nodes with an orphan pool size of 500

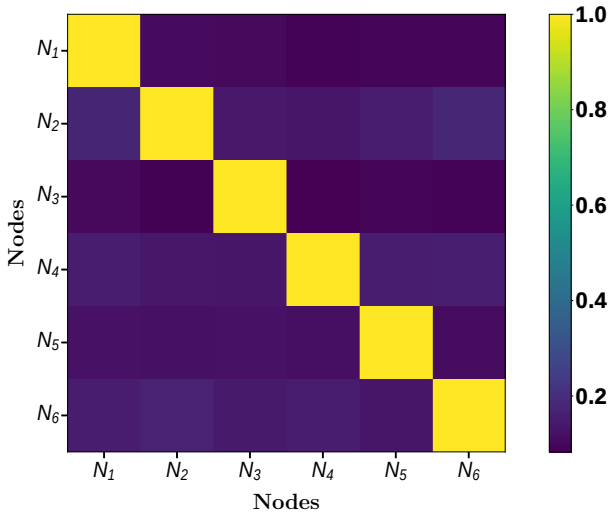

Fig. 9: Similarity matrix depicting average number of common peers across nodes during the first round of measurement period.

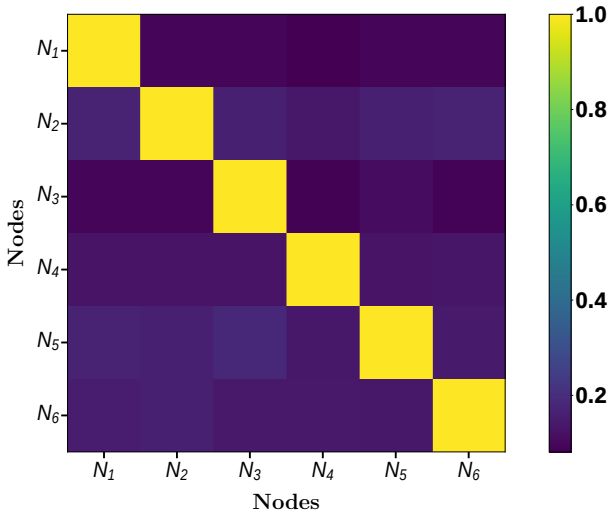

Fig. 10: Similarity matrix depicting average number of common peers across nodes during the second round of measurement period.

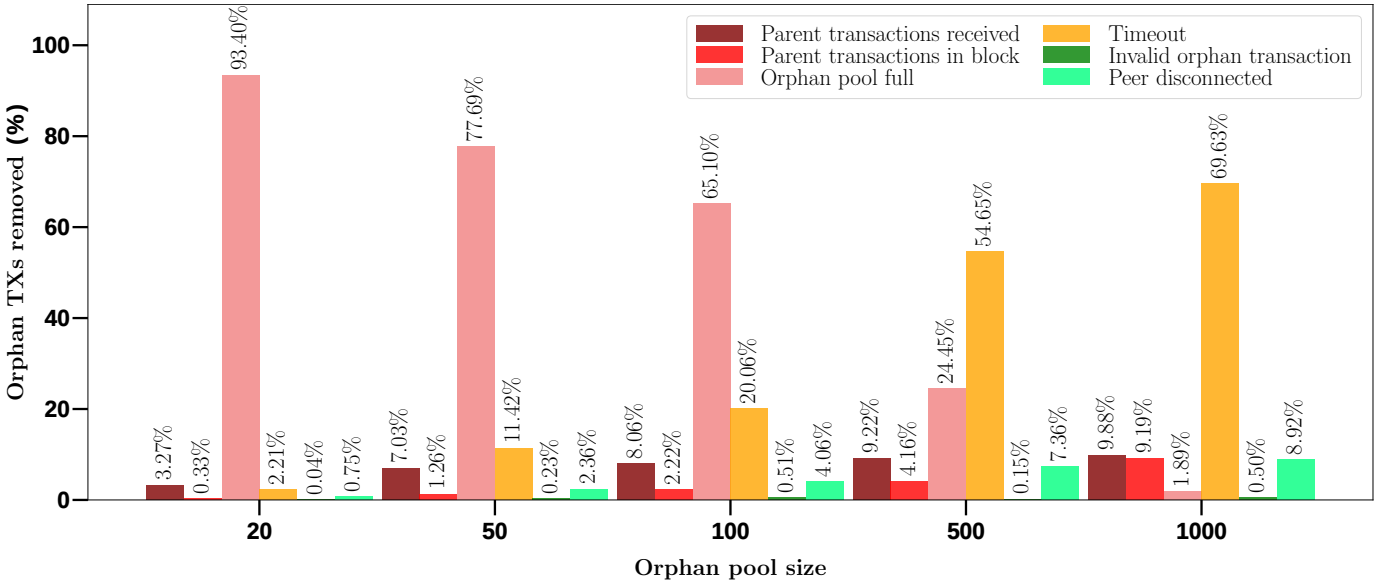

Fig. 11: Fraction of orphan transactions that are removed from the orphan pool due to each of the six causes across all nodes, under different pool sizes.

transactions, since smaller sizes lead to a large fraction of evictions as discussed in subsection IV-B.

B. Removal of orphan transactions from orphan pool

As specified in subsection II-A, there are six different cases in which a transaction is removed from the orphan pool. In this section, we analyze the fraction of orphan transactions that are removed from the orphan pool in each case.

Specifically, Fig. 11 shows the fraction of transactions 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. In fact, one of the nodes configured with an orphan pool of size 1000 (*i.e.,* node N_6) has *no* transactions evicted from the orphan pool, indicating that the pool never becomes full.

The remaining four cases contribute very little to the transaction being removed from the orphan pool. Of these, the major case that of transaction eviction from the orphan pool,

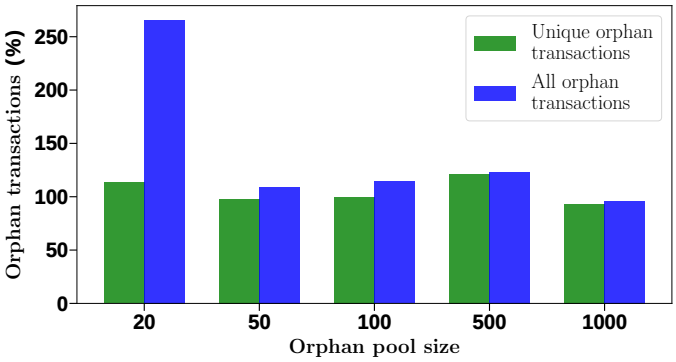

Fig. 12: Number of unique and total number of orphan transactions received across nodes with varying orphan pool sizes.

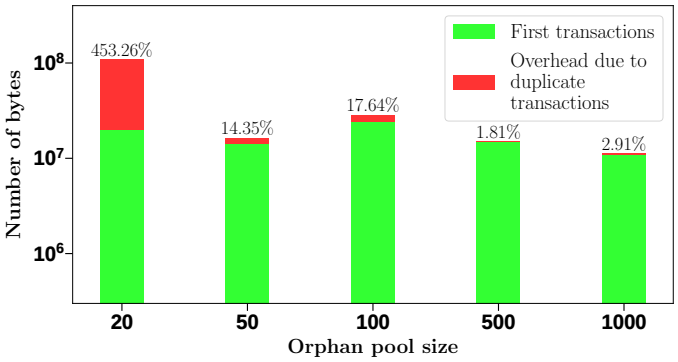

Fig. 13: Network overhead incurred by nodes with varying orphan pool sizes across nodes.

across nodes, is that the node receives the missing parent it had requested from its peers. Fig. 11 shows that as the size of the orphan pool increases, the fraction of orphan transactions that receive their respective missing parents gradually increases.

C. Addition of orphan transactions to orphan pool

In the previous section, we showed 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 II-A), 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. In this section, we specifically look at the number of times a transaction may be added to the orphan pool with varying orphan pool sizes.

To this end, the left bar in each column of Fig. 12 shows the *unique* transactions added to the orphan pools with varying sizes. The right bar of the respective column shows the *total* transactions added to the orphan pools with varying sizes. All values are normalized to the average number of *unique* transactions added to the orphan pools with a default size of 100 over the measurement period which, on average, is 5.72×10^4 transactions.

We observe yet another trend: 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 - a peer may re-announce a transaction that was previously removed from the orphan pool. Because the node does not have the transaction in either its mempool or the orphan pool, it accepts the transaction again to its orphan pool.

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. In the next section, we explain why multiple additions may pose a problem for network efficiency.

D. Network overhead

We next estimate the network overhead (*i.e.,* the number of bytes received) caused by receiving duplicate orphan transactions from peers. In our experiments, each time an orphan transaction is received, we add the size of the transaction: 32 bytes for the transaction hash in the `inv` message [40] and 32 bytes for the transaction hash in the `getdata` message [41]. Note that this provides a lower bound for the number of bytes transmitted each time a transaction is received, as the `inv` and `getdata` messages contain other fields, the total size of which would depend on the number of transactions packed in each message. We do not include this size in our calculation for simplicity. Similarly, we do not include the transport layer overhead in our estimation.

Fig. 13 shows statistics on the network overhead for duplicate orphan transactions received for the varying orphan pool sizes. The lower part of the stacked bar in each column shows the total number of bytes that are received when all *unique* orphan transactions are received for the first time. The upper part of the stacked bar in the respective column shows total number of bytes received when duplicates of the orphan transactions are received; note that the Y-axis in this figure is logarithmic. We also provide the cost of receiving duplicate orphan transactions (above each bar) as a fraction of the cost of receiving each orphan transaction for the respective orphan pool size. Assuming that the arrivals of orphan transactions is evenly distributed over the measurement period, an orphan pool size of 20 translates to an average rate of 1.32 `kbps` of transaction data in Fig. 13. On the other hand, for an orphan pool size of 1000, the average arrival rate of orphan transactions translates to only about 0.13 `kbps` of transaction data.

From the figures, we see 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 (cf. subsection IV-C).

Nodes	Round 1		Round 2	
	Add (%)	Remove (%)	Add (%)	Remove (%)
N_1	18.23	17.26	18.71	16.90
N_2	15.26	13.53	15.86	13.36
N_3	18.67	18.61	18.37	17.36
N_4	16.37	13.86	15.95	13.33
N_5	18.22	17.90	18.67	17.58
N_6	15.85	13.31	16.84	13.94

TABLE I: Average CPU usage of nodes with different orphan pool sizes.

E. Performance overhead

Finally, we explore the CPU and memory overhead incurred by varying orphan pool sizes. We empirically measure the CPU overhead with data from Unix `procfs`, and approximate the memory overhead. Our analysis shows that larger orphan pool sizes do not incur notable overhead for our node systems.

1) CPU overhead: The CPU overhead is observed by recording the CPU usage of the Bitcoin process every time an orphan transaction is added or removed from the orphan pool. TABLE I shows the *average* CPU usage of the Bitcoin process over the measurement period. The table shows that the difference in the average CPU usage of the Bitcoin process is barely distinguishable among the various orphan pool sizes. We attribute this to the data structure used for the orphan pool: relevant `std::map` operations typically have worst-case logarithmic time complexity [42]–[44].

2) Memory overhead: The Bitcoin core maintains three data structures related to orphan transactions. The first data structure represents the orphan pool. Each entry for an orphan transaction in the orphan pool contains i) the hash of the transaction (32 bytes), ii) a pointer to the actual transaction (16-byte integer on 64-bit architecture; 8-byte integer on 32-bit architecture; the size of this pointer is double that of an ordinary pointer because a `std::shared_ptr` is made of 2 pointers [45], iii) the ID of the peer that sent the transaction (8-byte integer), iv) expiration time of the transaction (8-byte integer), and v) position of orphan transaction in the orphan pool (8-byte integer on 64-bit architecture; 4-byte on 32-bit architecture).

Considering that the transaction would be stored in the mempool anyway if it were not an orphan, each orphan transaction incurs a memory overhead of 72 bytes on a 64-bit architecture, and 60 bytes on a 32-bit architecture.

The second data structure is used to maintain links between a missing parent and all orphan transactions that may spend from it. This efficiently resolves orphan status of all orphan transactions that depend on a missing parent once the latter is received from peers.

Each entry in this data structure contains i) the hash of the parent (32 bytes), ii) the index of the parent in the orphan transaction (4 bytes), and iii) a pointer to the orphan transaction in the orphan pool (8-byte integer on 64-bit architecture; 4-byte integer on 32-bit architecture). That is, each entry in this data structure takes up $36+8\times N$ bytes on a 64-bit architecture, and $36+4\times N$ bytes on a 32-bit architecture, where N is the number of all orphan transactions that spend from a missing parent.

It is tricky to theoretically justify a hard bound on the overhead incurred by this data structure. A transaction may spend from an arbitrary number of parents, an unknown number of which may be missing. Furthermore, not all parents may be missing at the same time, *i.e.,* a peer may not respond with *all* requested missing parents at the same time. On the other hand, an arbitrary number of orphan transactions may spend from the same missing parent.

Our empirical data, however, suggests that, orphan transactions across nodes with the varying orphan pool sizes have, on average, between 1 and 4 missing parents. where transactions across nodes with smaller pool sizes miss more parents; transactions across nodes with larger orphan pool sizes are very unlikely to miss more than 1 parent. Indeed, more than 90% of orphan transactions received by nodes configured with an orphan pool of size 1000 miss only 1 parent.

Similarly, across nodes with varying orphan pool sizes, the number of missing parents that orphan transactions share is in the range $(0, 1)$ on average. For every node, more than 98% of all orphan transactions received by that node share no parent.

Finally, for efficient random eviction of transactions from the orphan pool when the pool is full, a list is maintained. Each entry in the list is a pointer to a transaction in the orphan pool, with an overhead of 8-bytes for a 64-bit architecture and 4-bytes for a 32-bit architecture.

Consider, for example, a node configured with an orphan pool of size 1000 on a 64-bit architecture. This configuration incurs an average memory overhead of roughly 72 KB for the first data structure, 44 KB for the second data structure, and 8 KB for the third data structure for an aggregated average overhead of 122 KB, several orders of magnitude smaller than the typical memory on a modern system.

F. Varying orphan transaction timeouts

The findings in subsection IV-B show that as one increases the size of the orphan pool, transactions get primarily evicted due to timeouts. A natural question is whether changing the timeout from the default value of 20 minutes may help improve performance, and in particular the recovery of missing parents. Toward this end, we next present experimental results to evaluate the impact of varying timeouts.

Fig. 14 depicts the fraction of transactions removed from the orphan pool for each of the six cases specified in subsection II-A and different timeouts.

Increasing the timeout beyond the default of 20 minutes does not appear to decidedly improve performance. Specifically, the faction of transactions for which the parent transactions are recovered is 8.14% for the default timeout of 20 minutes, 5.81% for a timeout of 30 minutes, and 10.63% for a timeout of 60 minutes. On the other hand, reducing the timeout degrades performance (*i.e.,* 5.60% for a timeout of 15 minutes and 4.03% for a timeout of 10 minutes). Thus, the default timeout of 20 minutes appears appropriate.

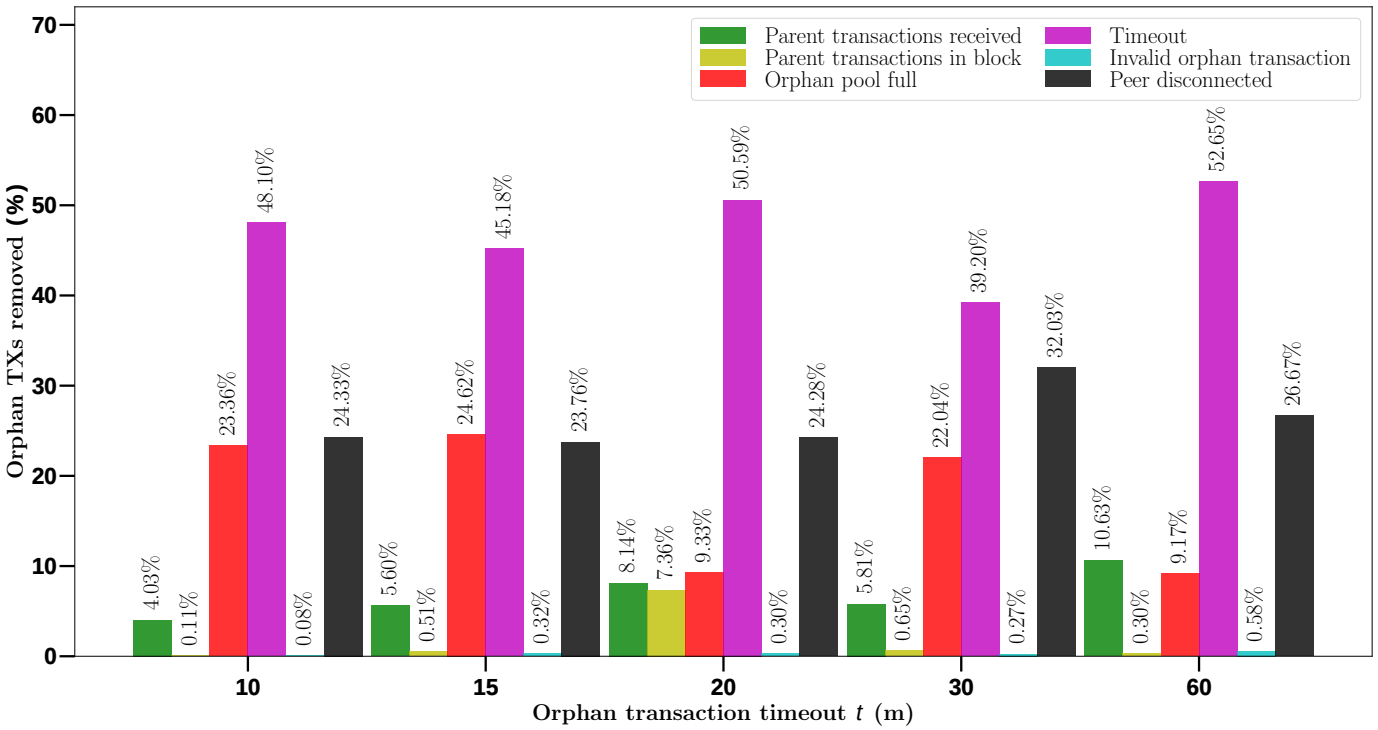

Fig. 14: Fraction of orphan transactions that are removed from the orphan pool due to each of the six causes across all nodes, under different timeouts.

V. 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. In this section, we analyze how an empty or stale mempool affects orphan transactions. We first describe our experimental setup and then present results.

A. Measurement setup

We configure three nodes with the same identical hardware and software specifications as described in subsection III-A. The nodes are configured with the default orphan pool size of 100 transactions and the default orphan transaction timeout of 20 minutes. To emulate the behavior of a node that has just joined the Bitcoin network with an empty or stale mempool, we clear the mempool of the nodes every 12 hours. The experiment runs from July 6, 2020 3:00 PM for two weeks. That is, we collect and present results obtained from data gathered over 84 session that are 12 hours long.

B. Fraction of orphan transactions

We first measure the fraction of incoming transactions that become orphan. We divide each of the 12 hour sessions into bins of 5 minute intervals. For each bin, we calculate the fraction of transactions that became orphan among all incoming transactions.

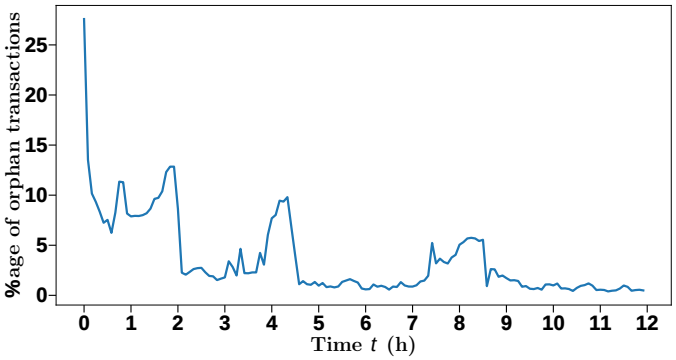

Fig. 15: Percentage of transactions that become orphan during each 5 minute bin interval of the 12 hour long sessions.

Fig. 15 shows, on an aggregate level, the fraction of transactions that became orphan during each bin's interval for the entire 12 hour session. We observe that when a node starts up, a large fraction of transactions (*i.e.,* above 25%) are added to the orphan pool. As the nodes stay connected, the fraction of orphan transactions drops, with occasional upward surges which can be attributed to the unsteady stream of incoming transactions as shown in Fig. 16.

We note that a node has fewer peers when it starts up as compared to in steady-state. Therefore, it receives a relatively smaller number of transactions at the beginning, as shown in Fig. 16.

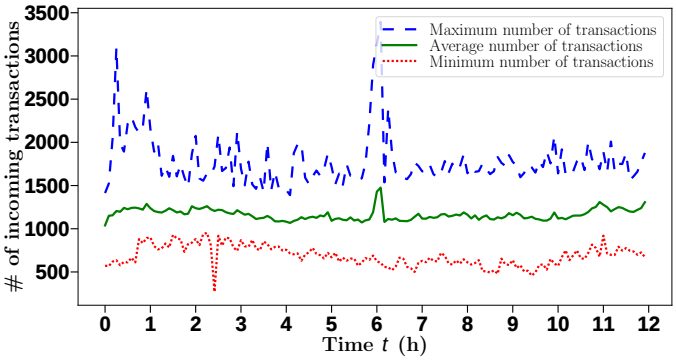

Fig. 16: Maximum, average, and minimum number of transactions received by nodes during each 5 minute bin interval of the 12 hour long sessions aggregated over all 84 sessions. The differences between the curves indicate that the number of incoming transactions varies across sessions.

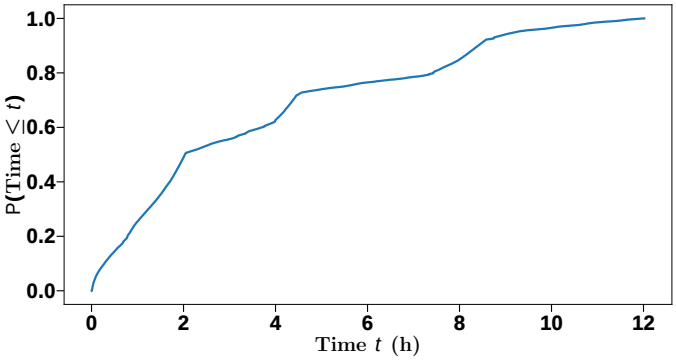

Fig. 17: CDF of arrival times of orphan transactions during measurement periods. Roughly 50% of all orphan transactions are received in the first two hours.

C. Arrival times of orphan transactions

We next analyze when during the measurement period orphan transactions are added to the orphan pool. For this purpose, we characterize the arrival times of orphan transactions (*i.e.,* the time at which an incoming transaction is deemed orphan and added to the orphan pool). The results are averaged over the 84 sessions, each of which is 12 hours long.

Fig. 17 shows, on an aggregate level, the CDF of the arrival times. We observe that the majority of orphan transactions arrive soon after a node starts up. Indeed, on average, roughly 50% of orphan transactions arrive within the first two hours of the 12 hours measurement sessions.

D. Removal of orphan transactions from orphan pool

Finally, we present an analysis of the causes of removal of transactions from the orphan pool, after a node joins the network. We note from the previous section that a large fraction of orphan transactions arrive within the first two hours. Hence, we zoom into this time frame and examine how each of the six scenarios (cf. subsection II-A) contributes to transactions being removed from the orphan pool.

Fig. 18 shows, on an aggregate level, the fraction of transactions that are removed from the orphan pool within the first two hours after a node joins the Bitcoin network. By design, when a node boots up, its orphan pool is empty. Therefore, we observe that immediately after a node joins the network, transactions are not removed from the orphan pool because the pool becomes full. Instead, a larger fraction of transactions are removed from the orphan pool because their missing parents are found. However, as the number of orphan transactions increases, the orphan pool fills up and evictions due to a full orphan pool become the leading reason for transactions being removed from the orphan pool. The remaining five causes contributes relatively little to the eviction of transactions from the orphan pool. These findings further confirm our earlier findings that Bitcoin nodes ought to operate with a larger orphan pool size to avoid unnecessary evictions and redundant network overhead (cf. subsection IV-D).

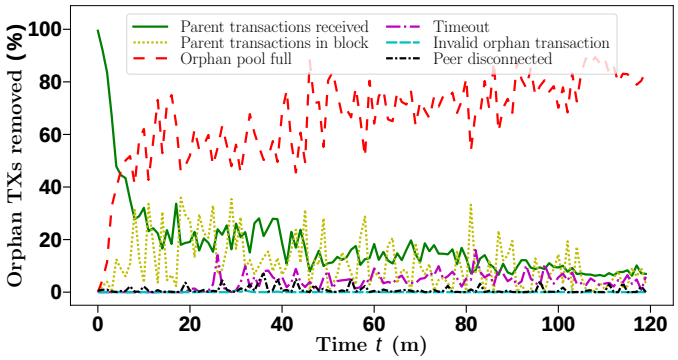

Fig. 18: Fraction of transactions that are removed from the orphan pool due to each of the six causes (cf. subsection II-A) over the first two hours of the 12 hour long sessions.

VI. Discussions and Limitations

We next discuss our results and point out some of their limitations.

Propagation of parent transactions. Recall that an orphan transaction is not propagated forward to other peers until all of its missing parents are found (cf. subsection II-A). One might then ask: why are there orphan transactions at all? Should not the peer that sent the orphan transaction to the measurement node have had the missing parents, or else it would not have forwarded the transaction to the measurement node? Answering this question, there are several reasons why a transaction forwarded by peers can end up in the orphan pool despite the peer having its missing parents.

First, results presented in Section V show that a large fraction of transactions that a node receives after joining the Bitcoin network become orphan. The peers of this node do not know in advance what transactions the node already has and, therefore, it is likely that the node will miss parents of transactions being announced by its peers. Since many Bitcoin nodes experience churn [34], [35], such scenarios are quite common.

In addition, each node maintains its own minimum acceptable fee which is a function of the node's configured mempool size and the amount of memory available. Any transaction received by the node that has a fee below this minimum is

rejected and is not added to the mempool and relayed to peers. A preliminary measurement shows that some transactions that are rejected due to low fee end up as missing parents of orphan transactions. We leave a detailed investigation to a future work.

Peer selection in measurement nodes. We focus on observing behavior of orphan transactions in regular, full Bitcoin nodes that participate in propagating information in the network but do not mine blocks. Our nodes discover and connect to peers on their own, similar to any regular, full Bitcoin node, so as not to introduce any unwanted bias in our data.

Performance impact of orphan transactions. We learned from Section V that nodes receive most orphan transactions during the first few hours of connecting to the network. As such, we can expect that a larger fraction of orphan transactions will arrive within these first few hours. This larger initial incoming traffic can be avoided by increasing the orphan pool size.

Ideas for future development. Our analysis shows that it is useful for nodes to configure a larger orphan pool size in order to reduce unnecessary network overhead. This may be especially beneficial for nodes that rejoin the network after a long downtime or that join the network for the first time.

Package Relay [46]–[48] is a proposed feature currently under discussion in the Bitcoin community. The goal of the feature is to package a transaction with all of its ancestors currently present in a node's mempool when relaying the transaction forward to its peers. It may be valuable to study whether this feature also helps reduce the number of transactions that become orphan.

VII. Conclusion

We have investigated circumstances under which a Bitcoin transaction is orphaned in Section III. Our data shows that orphan transactions have, on average, *fewer* parents than other transactions. The parents that cause transactions to become orphaned also have a lower transaction fee and a larger size relative to all received transactions. On an individual level, the missing parents also have, on average, a lower transaction fee per byte as compared to parents of all received transactions. This information can be utilized by Bitcoin users to appropriately set their own transaction fees and facilitate propagation through the network.

We have also documented the network and performance overhead incurred by orphan transactions for orphan pools of varying sizes. Our analysis in Section IV reveals that as the orphan pool size grows, more transactions are removed from the pool, not because the pool is full but because the transactions timeout. This in turn reduces the duplicate addition of transactions to the orphan pool, resulting in a much smaller network overhead. Our evaluations show that the performance overhead incurred by a larger orphan pool is insignificant, and it is thus advisable to set a larger orphan pool of larger size. On the other hand, changing the orphan transaction *timeout* from the default of 20 minutes does not appear to help. Indeed, increasing the default orphan transaction timeout does not decidedly improve performance, and reducing the timeout degrades performance.

We have also investigated the transient behavior of orphan transactions in nodes that join the network for the first time or after a long downtime (*i.e.,* they do not contain useful transactions in their mempools) in Section V. Our analysis shows that immediately after a node joins the network, on average, over 25% of the received transactions become orphans. Furthermore, over the measurement period, a large fraction of the orphan transactions, *i.e.,* roughly 50%, are added to the orphan pool during the first two hours. We also observe that, when a node first starts up with an empty orphan pool, most transactions are removed from the orphan pool due to reception of their missing parents. However, after a few minutes, an overflow of the orphan pool becomes the primary cause for the removal of orphan transactions. This finding further confirms the inadequacy of the default orphan pool size that is limited to 100 transactions.

Finally, our measurements show that missing parents are sometimes found many hours after their child transaction became orphan. We conjecture that this large delay may be caused by the Replace-by-Fee (RBF) feature [30] of Bitcoin. Another important finding is that in many cases, one or more missing parents are included in the same block as the orphan transaction. Since orphan transactions are not propagated, this may slow down the block propagation process (due to potential failure of compact blocks [34], [35]). Detailed investigation of these phenomena represent interesting areas for future work.

Acknowledgment

The authors would also like to acknowledge Sean Brandenburg for help with forward porting the log-to-file system to the newer version of the Bitcoin software.

References

[1] M. A. Imtiaz, D. Starobinski, and A. Trachtenberg, "Characterizing orphan transactions in the bitcoin network," in *2020 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, IEEE, 2020.

[2] "Bitcoin price, charts, market cap, and other metrics." `https://coinmarketcap.com/currencies/bitcoin/`. Online; Accessed: December 8, 2019.

[3] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system." `https://bitcoin.org/bitcoin.pdf`, 2009. Online.

[4] "The complete guide to Bitcoin fees." `https://99bitcoins.com/bitcoin/fees/`. Online; Accessed: December 8, 2019.

[5] "Confirmed transactions per day." `https://www.blockchain.com/en/charts/n-transactions?timespan=180days`. Online; Accessed: December 8, 2019.

[6] "Total number of transactions." `https://www.blockchain.com/charts/n-transactions-total?timespan=all`. Online; Accessed: December 8, 2019.

[7] D. Ron and A. Shamir, "Quantitative analysis of the full Bitcoin transaction graph," in *International Conference on Financial Cryptography and Data Security*, pp. 6–24, Springer, 2013.

[8] M. Ober, S. Katzenbeisser, and K. Hamacher, "Structure and anonymity of the Bitcoin transaction graph," *Future internet*, vol. 5, no. 2, pp. 237–250, 2013.

[9] M. Fleder, M. S. Kester, and S. Pillai, "Bitcoin transaction graph analysis," *arXiv preprint arXiv:1502.01657*, 2015.

[10] G. Di Battista, V. Di Donato, M. Patrignani, M. Pizzonia, V. Roselli, and R. Tamassia, "Bitconeview: visualization of flows in the Bitcoin transaction graph," in *2015 IEEE Symposium on Visualization for Cyber Security (VizSec)*, pp. 1–8, IEEE, 2015.

[11] M. Möser, R. Böhme, and D. Breuker, "An inquiry into money laundering tools in the Bitcoin ecosystem," in *2013 APWG eCrime Researchers Summit*, pp. 1–14, Ieee, 2013.

[12] A. Greaves and B. Au, "Using the Bitcoin transaction graph to predict the price of Bitcoin," *No Data*, 2015.

[13] D. McGinn, D. Birch, D. Akroyd, M. Molina-Solana, Y. Guo, and W. J. Knottenbelt, "Visualizing dynamic Bitcoin transaction patterns," *Big data*, vol. 4, no. 2, pp. 109–119, 2016.

[14] E. Androulaki, G. O. Karame, M. Roeschlin, T. Scherer, and S. Capkun, "Evaluating user privacy in Bitcoin," in *International Conference on Financial Cryptography and Data Security*, pp. 34–51, Springer, 2013.

[15] T. Ruffing and P. Moreno-Sanchez, "Valueshuffle: Mixing confidential transactions for comprehensive transaction privacy in Bitcoin," in *International Conference on Financial Cryptography and Data Security*, pp. 133–154, Springer, 2017.

[16] J. Herrera-Joancomartí and C. Pérez-Solà, "Privacy in Bitcoin transactions: new challenges from blockchain scalability solutions," in *International Conference on Modeling Decisions for Artificial Intelligence*, pp. 26–44, Springer, 2016.

[17] Q. Wang, B. Qin, J. Hu, and F. Xiao, "Preserving transaction privacy in Bitcoin," *Future Generation Computer Systems*, 2017.

[18] Y. Liu, X. Liu, C. Tang, J. Wang, and L. Zhang, "Unlinkable coin mixing scheme for transaction privacy enhancement of Bitcoin," *IEEE Access*, vol. 6, pp. 23261–23270, 2018.

[19] S. Meiklejohn and R. Mercer, "Möbius: Trustless tumbling for transaction privacy," *Proceedings on Privacy Enhancing Technologies*, vol. 2018, no. 2, pp. 105–121, 2018.

[20] Y. Kawase and S. Kasahara, "Transaction-confirmation time for Bitcoin: a queueing analytical approach to blockchain mechanism," in *International Conference on Queueing Theory and Network Applications*, pp. 75–88, Springer, 2017.

[21] Y. Sompolinsky and A. Zohar, "Accelerating Bitcoin's transaction processing. fast money grows on trees, not chains.," *IACR Cryptology ePrint Archive*, vol. 2013, no. 881, 2013.

[22] S. Kasahara and J. Kawahara, "Effect of Bitcoin fee on transaction-confirmation process," *Journal of Industrial & Management Optimization*, vol. 15, no. 1, pp. 365–386, 2019.

[23] Y. Zhu, R. Guo, G. Gan, and W.-T. Tsai, "Interactive incontestable signature for transactions confirmation in Bitcoin blockchain," in *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*, vol. 1, pp. 443–448, IEEE, 2016.

[24] S. Delgado-Segura, S. Bakshi, C. Pérez-Solà, J. Litton, A. Pachulski, A. Miller, and B. Bhattacharjee, "Txprobe: Discovering Bitcoin's network topology using orphan transactions," in *International Conference on Financial Cryptography and Data Security*, pp. 550–566, Springer, 2019.

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

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

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

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

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

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

[31] "Bitcoin orphan transactions and cve-2012-3789." `https://cryptoservices.github.io/fde/2018/12/14/bitcoin-orphan-TX-CVE.html`. Online; Accessed: February 12, 2020.

[32] "Dos fix for maporphantransactions." `https://github.com/bitcoin/bitcoin/pull/911`. Online; Accessed: February 12, 2020.

[33] "bitcoin-releases." `https://github.com/nislab/bitcoin-releases/tree/icbc2020`. Online; Accessed: February 13, 2020.

[34] M. A. Imtiaz, D. Starobinski, A. Trachtenberg, and N. Younis, "Churn in the Bitcoin Network: Characterization and impact," in *2019 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, pp. 431–439, IEEE, 2019.

[35] M. A. Imtiaz, D. Starobinski, A. Trachtenberg, and N. Younis, "Churn in the bitcoin network," *IEEE Transactions on Network and Service Management*, pp. 1–1, 2021.

[36] "A practical guide to accidental low fee transactions." `https://hackernoon.com/holy-cow-i-sent-a-bitcoin-transaction-with-too-low-fees-are-my-coins-lost-forever-7a865e2e45ba`. Online; Accessed: December 3, 2019.

[37] "Is there any max limit of a mempool?." `https://bitcointalk.org/index.php?topic=1714006.msg17171748#msg17171748`. Online; Accessed: December 5, 2019.

[38] S. Jiang and J. Wu, "Bitcoin mining with transaction fees: a game on the block size," in *2019 IEEE International Conference on Blockchain (Blockchain)*, pp. 107–115, IEEE, 2019.

[39] "bitcoin-logs." `https://github.com/nislab/bitcoin-logs/tree/icbc2020`. Online; Accessed: February 13, 2020.

[40] "Protocol documentation (inv)." `https://en.bitcoin.it/wiki/Protocol_documentation#inv`. Online; Accessed: December 3, 2019.

[41] "Protocol documentation (getdata)." `https://en.bitcoin.it/wiki/Protocol_documentation#getdata`. Online; Accessed: December 3, 2019.

[42] "`std::map::erase`." `http://www.cplusplus.com/reference/map/map/emplace`. Online; Accessed: December 4, 2019.

[43] "`std::map::count`." `http://www.cplusplus.com/reference/map/map/count/`. Online; Accessed: December 4, 2019.

[44] "`std::map::erase`." `http://www.cplusplus.com/reference/map/map/erase/`. Online; Accessed: December 4, 2019.

[45] "Exploring `std::shared_ptr`." `https://shaharmike.com/cpp/shared-ptr/`. Online; Accessed: December 5, 2019.

[46] "Package relay." `https://bitcoinops.org/en/topics/package-relay/`. Online; Accessed: November 25, 2020.

[47] "Package relay design questions." `https://github.com/bitcoin/bitcoin/issues/14895`. Online; Accessed: November 25, 2020.

[48] "Transaction package relay." `https://gist.github.com/sdaftuar/8756699bfcad4d3806ba9f3396d4e66a`. Online; Accessed: December 4, 2020.

Muhammad Anas Imtiaz (Graduate Student Member, IEEE) received the B.S. degree *(cum laude)* in computer engineering from the National University of Computer and Emerging Sciences, Lahore, Pakistan, in 2014, with a silver medal. He is currently pursuing the Ph.D. degree in computer engineering with Boston University (BU), where he joined in 2017. He works as a Doctoral Research Fellow in the NISLab at BU. Prior to joining BU, he worked as a Software Development Engineer from 2014 to 2016, and as a Senior Software Development Engineer from 2016 to 2017 at Mentor Graphics (now Siemens EDA). At Mentor, his responsibilities included development, maintenance, and upgrade of several automotive software and legacy products offered by the company. His research investigates the presence and effects of churn in the Bitcoin network, and possible improvements in the Bitcoin protocol to help mitigate such issues. His work on orphan transactions in the Bitcoin network won a Best Paper Award at the IEEE ICBC 2020 conference. He participated as a reviewer at IEEE TNSM in 2020, and as a Shadow PC at the AMC IMC 2019 conference.

David Starobinski (Senior Member, IEEE) received the Ph.D. degree in electrical engineering from the Technion-Israel Institute of Technology in 1999. He is a Professor of Electrical and Computer Engineering, Systems Engineering, and Computer Science at the Boston University. He was a Visiting Postdoctoral Researcher with the EECS Department, UC Berkeley from 1999 to 2000, an invited Professor with EPFL from 2007 to 2008, and a Faculty Fellow with the U.S. DoT Volpe National Transportation Systems Center from 2014 to 2019. His research interests are in cybersecurity, wireless networking, and network economics. He received the CAREER Award from the U.S. National Science Foundation in 2002, the Early Career Principal Investigator Award from the U.S. Department of Energy in 2004, the BU ECE Faculty Teaching Awards in 2010 and 2020, and the Best Paper Awards at the WiOpt 2010, IEEE CNS 2016, and IEEE ICBC 2020 conferences. He is on the editorial board of the IEEE Open Journal of the Communications Society and was on the editorial boards of the IEEE Transactions on Information Forensics and Security, and the IEEE/ACM Transactions on Networking.

Ari Trachtenberg received the S.B. degree from MIT in 1994, and the M.S. and Ph.D. degrees in computer science from the University of Illionis Urbana-Champaign (UIUC) in 1996 and 2000, respectively. He is a Professor of Electrical and Computer Engineering, Computer Science, and Systems Engineering with Boston University (BU), where he has been since September, 2000. He has also been a Distinguished Scientist Visitor with Ben Gurion University, a Visiting Professor with the Technion-Israel Institute of Technology, and worked with TripAdvisor, MIT Lincoln Lab, HP Labs, and John Hopkins Center for Talented Youth. His research interests include cybersecurity (smartphones, offensive and defensive), networking (security, sensors, localization), algorithms (data synchronization, file edits, file sharing) and error-correcting codes (rateless coding, feedback). He has been awarded the ECE Teaching Awards from BU in 2003 and 2013, a Kern Fellowship from BU in 2012, the NSF CAREER from BU in 2002, and the Kuck Outstanding Thesis from UIUC in 2000.