On Availability for Blockchain-Based Systems

Ingo Weber*†, Vincent Gramoli‡*, Alex Ponomarev*,
Mark Staples*†, Ralph Holz‡*, An Binh Tran*, Paul Rimba*
*Data61, CSIRO, Sydney, Australia
Email: {firstname.lastname}@data61.csiro.au
‡School of Information Technologies, University of Sydney, Australia
†School of Computer Science and Engineering, University of New South Wales, Sydney, Australia

***Abstract*—Blockchain has recently gained momentum. Startups, enterprises, banks, and government agencies around the world are exploring the use of blockchain for broad applications including public registries, supply chains, health records, and voting. Dependability properties, like availability, are critical for many of these applications, but the guarantees offered by the blockchain technology remain unclear, especially from an application perspective. In this paper, we identify the availability limitations of two mainstream blockchains, Ethereum and Bitcoin. We demonstrate that while read availability of blockchains is typically high, write availability—for transaction management—is actually low. For Ethereum, we collected 6 million transactions over a period of 97 days. First, we measured the time for transactions to commit as required by the applications. Second, we observed that some transactions never commit, due to the inherent blockchain design. Third and perhaps even more dramatically, we identify the consequences of the lack of built-in options for explicit abort or retry that can maintain the application in an uncertain state, where transactions remain pending (neither aborted nor committed) for an unknown duration. Finally we propose techniques to mitigate the availability limitations of existing blockchains, and experimentally test the efficacy of these techniques.**

I. Introduction

Blockchains are a new kind of replicated database (a 'distributed ledger') which can be operated without the control of any single party. Blockchain technology emerged to support the Bitcoin cryptocurrency [1]. A second generation of blockchains are more general-purpose: transactions can record data about any kind of application domain, and can deploy and execute user-defined scripts ('smart contracts'). This greatly expands the potential uses for blockchain technology. Many startups, enterprises, banks, and governments are exploring its applications in areas as diverse as electronic health records, voting, energy supply, and protecting critical civil infrastructure [2]. These applications typically involve cross-organizational business processes [3], taking advantage of the neutral ground provided by a blockchain. Many of these applications have critical dependability requirements, even beyond those driven by the size of trade in the underlying cryptocurrencies (millions of US dollars per day[1]).

Understanding the dependability properties supported by blockchains has become crucial. Prominent blockchain systems, specifically those using Nakamoto consensus [1], can only offer probabilistic guarantees to their clients in terms of the immutability of recorded transactions. Misinterpreting these guarantees or underestimating the time required for a transaction to become immutable may lead to failures in dependent applications. This poses a significant problem for applications that rely on blockchain as a component, data store, or software connectors (as suggested by Xu *et al.* [4]).

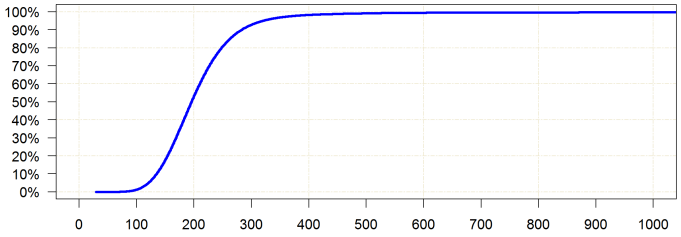

Figure 1: Time (in second) to commit transactions in Ethereum

To measure the availability of mainstream blockchains in terms of their responsiveness, we have first to define the notion of transaction commit needed by the applications running on top of these blockchains. Clients can initiate transactions as long as they are connected to active peers of the blockchain. Although necessary, the activity of some of these peers is not sufficient to guarantee the commit. A *commit* requires instead that some peers create a sequence of blocks, whose first one includes the transaction [5]. The length of the required sequence may vary but is used by blockchain applications to consider that the transaction has successfully executed. This number is typically 6 in Bitcoin and 12 in Ethereum[2].

The problem is that clients cannot expect a time for their transaction to commit, despite sufficiently many blocks being created every 3 minutes (resp. 1 hour) in expectation in Ethereum (resp. Bitcoin). For example, consider Fig. 1, which plots the cumulative distribution function (CDF) of the period between the time we observed transactions being announced and the time we observed them as committed (we detail our experiment in Section IV). During our experiment, most transactions (actually 61.5%) took actually more than 3 minutes to commit, 13.8% of the transactions were not committed after 4.5 minutes, *i.e.,* they experienced a delay of 50% of the target commit time. This variance in time to commit a transaction can lead to availability failures for client applications.

In this paper, we explore how mainstream proof-of-work blockchains impact the dependability of systems built upon them. We focus on the availability of functions that such systems need, and how they are adversely impacted by a number of factors. Our primary contributions are as follows.

[1] https://coinmarketcap.com/exchanges/volume/24-hour/

[2] https://ethereum.stackexchange.com/questions/183/how-should-i-handle-blockchain-forks-in-my-dapp/203#203.

We investigate which factors cause transactions to remain uncommitted for much longer than the blockchain design promises and previous work assumed. In particular, we find that network reordering plays a much more important role than previously assumed and can cause significant delays. For the Ethereum blockchain, we investigate how durable transaction inclusion in the chain really is, and what the impact of the user-defined variables 'gas price' and 'gas limit' is on transaction inclusion. We also investigate the effect of the so-called 'block gas limit'. Because we find that transactions can remain uncommitted and even 'stuck' in the transaction pool, we identify the need for a mechanism to abort transactions. As the current blockchain designs do not feature such a mechanism, we carry out experiments to simulate the same effect and determine under which conditions transactions can be aborted.

The remainder of this paper starts with an overview of blockchain background in Section II. The issue of transaction commit time is discussed in Section III for Bitcoin and in Section IV for Ethereum. The impact of network-defined parameters is analysed in Section V. Then, the problem of missing abort mechanisms for transactions is analysed and mechanisms to simulate transaction abort are explored in Section VI. We present related work in Section VII and conclude in Section VIII.

II. Background

Blockchains have risen to prominence in recent years with the introduction of cryptocurrencies. Bitcoin and Ethereum are the two first blockchain in terms of market capitalization. As a technology, however, they support a wider range of use cases. A blockchain system can be thought of as an append-only, public ledger that keeps track of transactions made by participants. In most cases, these transactions relate to some (virtual) asset, and often involve moving quantities of the asset from one account to another.

Every participant in the blockchain system holds a local copy of the ledger and runs a network client that relays transactions to the entire network. The client can also inject new transactions into the network.

Transactions are signed using asymmetric cryptography: only the owner of a given private key can create and sign her transactions, but all participants can verify who signed a certain transaction using the corresponding public key. A Public key doubles as an address that belongs to the owner; it is freely propagated as transaction origins and destinations. However, an address is just a bitstring and does not typically reveal the true identity of the actual owner. In this sense, an address functions as a pseudonym. In general, a participant may also have any number of addresses, which can be grouped by locally-running software (the *wallet*) into *accounts*.

This design can ensure that all participants know what amount of the asset is associated with which address if an additional property holds: the public ledger is correctly synchronised between the participants, *i.e.,* all participants are in consensus about which transactions have been successfully made. A balance can then be computed for each address. If this is not guaranteed, it is trivial to 'double-spend' a balance as an attacker could strategically send his transaction to selected participants only.

To achieve the necessary consensus, transactions are grouped into blocks, which are cryptographically linked to form a linked list, the blockchain. Hence, consensus must be achieved on the content and order of the blocks. This is achieved by a blockchain's mechanism of *block creation*, which is commonly called *mining* or *solving a block*. The oldest and the most widely used form of mining is *proof-of-work*. Other forms (*e.g.,* proof-of-stake) exist, but are not investigated in this paper.

The proof-of-work mechanism used in consensus protocols works as follows. A *miner* (participant who wishes to create a block) groups transactions into a block, adds a random number of her choosing, and computes a hash sum over this block. For a block to be accepted by the network, this hash sum is interpreted as an integer, which must be smaller than a certain target value. The target value depends only on the previous block's hash sum and deterministic factors; it can be computed independently by each participant. The challenge for the miner is to find a random number that will yield a hash sum with the desired properties. The hash functions that are used in blockchain systems are cryptographically secure hash functions. Hence, there is no known better way of finding a target value than brute-force search. Consequently, with the target values ever decreasing, miners must invest more computational power. Once a miner finds such a target value, it is allowed to award itself a certain amount of the asset for having invested computational resources (the exact amount is deterministic and often a function of the length of the blockchain). This is also included in the block as a transaction and acts as an incentive to participate. The miner then signs the block and propagates it to the network, which may accept it and consider it as the latest state of the blockchain. In addition to the block reward, participants can add additional incentive for miners by adding a self-chosen *transaction fee* that miners who include the transaction in a mined block are allowed to claim.

Although it is very unlikely, two miners may occasionally find a new block at the same time and propagate it. This is called a *fork*, which means the blockchain is not in consensus and participants must wait for the next block (which chooses one of the two candidates as its predecessor) to re-establish consensus. The key strategy (Nakamoto consensus [1]) to resolve the fork is that every miner will always work from the currently longest chain. By a statistical argument, one can show that the more blocks that have been found since the inclusion of a transaction in a block, the less likely a new fork which produces an even longer blockchain that is accepted by the network will happen. For Bitcoin, the commonly agreed value is to wait for 6 blocks; for Ethereum it is 12. The appropriate number of confirmation blocks for a particular application depends on the value of the transactions, the cost (computational power) of mining blocks, and threat of hostile attack. Sufficient confirmations mean that it is sufficiently difficult for an attacker to mount enough computational power to find a longer sequence of blocks to replace the current consensus (and hence tamper with it). The well-known '51% attack' is possible when controlling strictly more than 50% of the network's combined computational resources, but more recent research has described strategies that work with less [6].

In a blockchain system with a large number of participants, the computational power required to solve a block first is nor-

mally out of reach for single participants, even with dedicated hardware. A common way to participate in a blockchain is to join a *mining pool*, where block creation is distributed over many individual participants. Any block reward is shared by participants. These mining pools were possibly not foreseen by Bitcoin's inventor. In fact, some attack strategies leverage the mining power that they consolidate, threatening dependability (especially integrity and availability).

Modern blockchains such as Ethereum provide remarkable new features, such as so-called smart contracts. These are programs stored on the chain and run by all participants. A pricing mechanism ('gas price') also allows contract authors to offer different prices for the execution of a contract. We return to this in Section IV.

III. Commit of Bitcoin Transactions

In this section, we explore the impact that affects Bitcoin commit-time and show that re-ordering of transactions play an active role.

A peculiarity of Bitcoin is the way transactions are linked: they transfer currency from a number of source addresses to a number of destination addresses. Transaction outputs become the inputs of new transactions. If the sum of the outputs is less than the sum of the inputs, this is interpreted as an additional output that pays a fee to the miner who mines the block containing this transaction. This acts as an incentive for miners. As a result, miners tend to optimize block creation by preferring transactions with higher fees. The transaction fee is often the only variable that client software asks Bitcoin users to choose consciously when creating a new transaction.

However, transactions can also experience delay due to other factors. An important one is that transactions must arrive (roughly) in-order, for a node (and the network) to be able to process them fast. Incoming transactions are handled by the so-called *mempool*. If the referenced input transactions (the 'parents') are yet unknown, a miner will delay the inclusion of the new transaction—it is then a so-called *orphan*. Miners may choose to keep orphans in the mempool while waiting for the parent transactions to arrive, but they may also expunge orphans after a time-out they choose. A second factor that could come into play, albeit one that only experienced users will set, are so-called locktimes: a transaction can contain a parameter declaring it invalid until the block with a certain sequence number has been mined. This makes it possible to set an 'execution date' for transactions.

Note that out-of-order arrival may be the result of a number of factors: the forwarding behaviour of a node depends on the implementation and is different even between versions of the 'official' Bitcoin Core client. It may naturally also depend on the load on miners (leading to low throughput as evidenced by an ongoing community discussion[3]). Transient connectivity issues and Internet routing constellations may also be at play[4]. Also note that transactions may be rejected by the mempool for certain reasons. We explain these below as we encounter them.

[3] https://en.bitcoin.it/wiki/Block_size_limit_controversy

[4] This is why projects such as Fibre (http://bitcoinfibre.org/public-network.html aim to provide high-speed links between certain locations.

	Experiment 1	**Experiment 2**
Start collection of TX	2016-11-29 20:25 UTC	2017-04-13 13:11 UTC
End collection of TX	2016-11-30 21:36 UTC	2017-04-14 14:15 UTC
No. of preceding block	441,177	441,476
No. of first block 24hrs after end of TX collection	461,721	462,003

Table I: Overview of experiment runs (TX: transaction)

A. Data Collection Methodology

We modified the `btcd` implementation of Bitcoin, which is fully compliant with Bitcoin's standard inclusion tests[5], and changed the mempool to log every incoming transaction together with the result of the applied checks. We collected incoming Bitcoin transactions and determined the time it took for them to be committed in the blockchain. We ran our experiment twice to allow for varying network conditions and growth of the network. Each experiment lasted ca. 25 hours; the first was conducted in November 2016, the second in April 2017. Table I provides the details. It should be noted that websites like https://blockchain.info/unconfirmed-transactions reported high network load while the second experiment was being carried out, with 25,000–30,000 transactions waiting for inclusion. Checks by the mempool yield results that fall into three categories. The first category are transactions that pass all tests ('straight-accept'). The second are rejected transactions. The third category are the 'orphaned' transactions, which reference the output of transactions that have themselves not been received yet, *i.e.,* a case of out-of-order arrival.

Our timing measurements are precise to the second. We removed all duplicate log entries, *i.e.,* a duplicate transaction result having arrived, and receiving the same result from the mempool checks. We only kept the earliest such transaction that had arrived. For example, if an orphan transaction arrives a second time while the parents have not arrived yet, this transaction would be filtered out (it has no effect on commit time). If the transaction arrives again, however, and is no orphan anymore because our node has also learned about the input transactions in the meantime, this is logged as a separate event. Concerning transactions being included in the blockchain, we defined an observation window from the first block preceding our transaction collection to the first block 24 hours after the end of our collection period. We counted which transactions were committed during this window (*i.e.,* inclusion in one block plus five subsequent blocks in the chain).

B. Inclusion Time of Bitcoin Transactions

We collected roughly 300,000 transactions in each experiment. Table II summarizes the associated statuses in the mempool. The two experiments show remarkably similar numbers for the total number of collected transactions, but the fraction of orphans in the second experiment was higher at 2.74% of the total number of transactions (0.87% in experiment 1).

There is also a striking difference in the number of transactions that were rejected because they had already been received ('already in mempool'). It seems plausible that the reported 'backlog' was causing additional forwarding and

[5] https://github.com/btcsuite/btcd

Status	Experiment 1	Experiment 2
Total collected TX	370,858	372,500
Straight accept	307,764	265,958
Orphan	2675	7274
(Reject: already in mempool)	55,706	90,695
(Reject: double-spend in mempool)	1339	296
(Reject: other)	2094	1882

Table II: Overview of collected transactions and status at arrival. Note that orphaned transactions may arrive a second time and get assigned a different status; hence the numbers in the table do not add up to the total number of collected transactions. We also group rarer rejections.

queuing on our node as well. The other reasons for rejection were much less frequent. We observed some attempts to double-spend a coin (*i.e.,* the same, unspent outputs were used as input in two different transactions)[6]. The other reasons are more arcane and grouped together in our table (*e.g.,* wrong format, failing certain sanity checks, etc.).

We summarize the commit times we determined in Table III. Note that they are significantly higher and more varied in the second experiment. Fig. 2 plots the commit times for the two forms of transactions that are our primary interest. The blue curves refer to transactions that were a 'straight-accept', *i.e.,* the parent transactions were known and the incoming transaction passed all mempool tests. The violet curves are the transactions that were orphans upon arrival.

In both experiments, orphans seem to be committed later than transactions that were directly accepted. However, the additional delay is much higher in the second experiment (where the network was under high load). In our first experiment, about 60% of orphans were included within the same time span as normal transactions. In fact, 31% of orphans took longer than 2 hours to be included, 21% longer than 3 hours, and 8% took longer than 5 hours. For directly accepted transactions, these values were slightly different: 17% of them took longer than 2 hours, 9.5% longer than 3, and 5% longer than 5 hours. In our second experiment, roughly 40% of orphans had similar commit times as directly accepted transactions. The majority experienced very significant delays: the median was almost 20% higher, and the third quantile is more than 2.5 times as high as that for straight-accepts. We also note that only 1.2% of orphans and 1.6% of directly accepted transactions had not been included by the end of our observation period in experiment 1. In experiment 2, more than 20% of orphans had not been included (but almost all straight-accept transactions).

Factors other than the out-of-order arrival might still exercise considerable influence on commit times. We hence decided to investigate two further factors: transaction fees and locktimes. We first determined the number of transactions with a zero fee. This was always very low: for the straight-accepts, it was 74 and 12 in experiment 1 and 2, respectively. The orphans *never* had a zero transaction fee. Fig. 3 shows a box plot of transactions fees with the zero values filtered out. We can see that transaction fees are considerably higher in the second experiment, but

[6]Note that these are not necessarily malicious attempts—they could also be attempts to abort a previous transaction. Instructions circulate on the Internet, *e.g.,* http://bitzuma.com/posts/how-to-clear-a-stuck-bitcoin-transaction/. We explain how to abort an Ethereum transaction in Section VI).

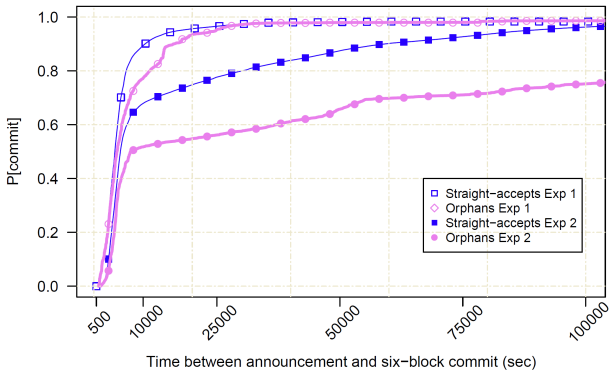

Figure 2: Time between reception of transaction and commit. Note the logarithmic x-axis.

Type	Min	Q1	Median	Mean	Q3	Max
Experiment 1:						
Orphans	944	3096	4635	7582	8334	117,585
Accepts	676	2887	4234	5475	5901	150,123
Experiment 2:						
Orphans	1293	4280	6337	34,912	51,352	174,516
Accepts	1165	3873	5364	18,417	19,286	171,566

Table III: Summary of commit time distributions (in sec) for orphans and straight-accepts during our experiments.

there is no difference between straight-accepts and orphans in experiment 1. In experiment 2, orphans even have slightly higher fees. It is very unlikely that lower transaction fees are a cause for delayed commit of orphans.

We extracted the locktimes for our collected transactions and the locktimes of their parents. As our logger had not captured the full content of transactions arriving in the mempool (but only hash value and timestamp), we conducted this analysis only for those transactions that had been incorporated into the blockchain. The vast majority of transactions had no locktime set: in experiment 1, only 15% of straight-accepts and 12% of orphans had a value that was not zero. In experiment 2, the numbers were 23% and 17%, respectively. While this may signal an increase in the use of the feature, orphans *never* had locktimes beyond the observation window. Orphans in experiment 1 had locktimes that ended at least 3 hours before the end of the observation window; in experiment 2 it was six hours. In contrast, straight-accepts did have locktimes that extended considerably beyond the end of the observation window. In experiment 1 and 2, nearly 100% of transactions also had locktimes similarly near the end of the observation window. However, we found some decidedly optimistic locktimes on the order of 1.5–1.7 billion (block sequence number). With 10 minutes being the average time between two Bitcoin blocks, these transactions cannot be included before the year 30.166. The obvious limitation of our work here is that we do not know the locktimes of those orphans that were not included in the blockchain by the end of our observation period. Given the above results, however, we still feel confident to say that locktimes are not likely to be a decisive factor in commit delay of orphans.

Naturally, there may still be confounding factors in our study that we could not control for in this experiment. For example, we do not have information about node connectivity outside of

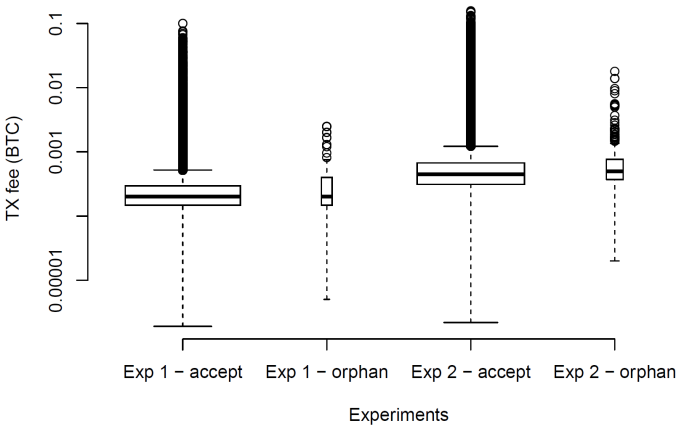

Figure 3: Box plot of transaction fees by transaction category. Note the logarithmic y-axis.

our observation post, Australia, and could not determine the (ever changing) Internet routing constellation that the Bitcoin network is exposed to. Note that propagation times in the Bitcoin network have been investigated before [7]. Our study suggests that it is worthwhile to revisit this topic.

IV. Commit of Ethereum Transactions

In this section, we first explain why Ethereum transactions are not guaranteed to be committed regardless of their validity, we then identify gas price, gas limit, and the network as factors that affect commit time.

A. Ethereum Transaction Handling

We first give an overview of Ethereum's interesting transaction handling. Fig. 4 captures the life cycle of individual transactions in the Ethereum blockchain. It starts with the submission of a transaction into the (virtual distributed) transaction pool across all miners. A transaction lifespan can be split into consecutive phases: (i) the announcement of the transaction in the system; (ii) the inclusion of the transaction in a newly mined block on some branch of the chain; (iii) the inclusion of the transaction in a block part of the main chain; and (iv) the commit of the transaction after sufficiently many blocks are subsequently mined.

There is no certainty whether a particular transaction will eventually be committed or whether it will be *outdated*, in that it will be considered an invalid transaction forever. Moreover, it is impossible to know whether a transaction that is invalid in some state of the system will never be valid in a later state. More specifically, the aforementioned step (ii) is not sufficient to guarantee that a transaction Tx is permanently added to the blockchain: if the blockchain forks, then the block comprising the transaction may simply be discarded, in which case it could be re-included later.

When a transaction is included in a block, it has been validated beforehand, *i.e.,* its digital signature has been checked, as well as the validity of parameters like the nonce (sequence number of transactions relative to a given source account), and that there are sufficient funds in the source account. If all blocks that included the transaction become part of a shorter chain than the main chain—they become so-called *uncles* in Ethereum terminology—then the transaction goes back into

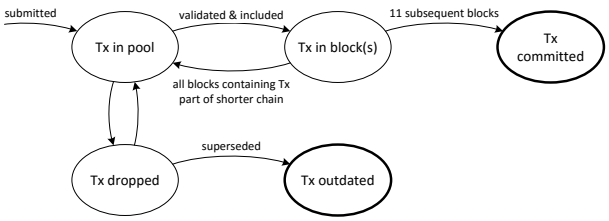

Figure 4: State machine for an individual transaction

the transaction pool. This may happen more than once and, theoretically, there is no upper limit. While the transaction is in the pool, it may also be dropped. This is a local decision of miners, and it is impossible for any node in the network to know with certainty that all miners have dropped the transaction. Only when the nonce of the transaction becomes outdated, *i.e.,* another transaction from the same source account with the same nonce got committed, can a node be certain that the transaction is invalid and will not be included in any valid block. Otherwise the transaction may resurface at a later point, and get included in the chain.

To determine with high probability that a transaction is permanently included in the blockchain, one has first to wait for several blocks to be mined after the first inclusion and refer to the block including the transaction of interest as an ancestor. Each of these subsequent blocks are called confirmations and when sufficiently many confirmations occurred after the transaction block inclusion, then the transaction is considered *committed*, *i.e.,* believed to remain in the blockchain forever with very high probability. The current version of Ethereum requires 11 blocks after the transaction inclusion for the transaction to be committed, but other values may be required by other versions [8].

B. Data Collection and Basic Statistics

We wrote an observation node for the Ethereum blockchain by modifying a client node to listen to the network and collect all transactions and block announcements. Our local client is based on the Ethereum implementation `geth` in version 1.5.3. We set the minimum gas price to 0 and otherwise use the default settings. One important change we made was to allow our client to connect to a maximum of 500 nodes instead of the default 25. This allowed it to collect a maximum of transactions—in general, it was communicating with more than 400 nodes at any time.

We modified the geth client such that all transaction announcements were intercepted before any verifications take place in order to prevent interesting transactions from being discarded. For each announced transaction, we recorded the local time of observation as transactions do not contain a timestamp. Similarly, for all incoming blocks we recorded the timestamp upon arrival. Although each block in Ethereum has a timestamp, the timestamp refers to the time where the mining of this block started and not the time when the block was generated. Moreover, the clocks of miners may not be perfectly synchronized.

We collected data over a 3.5 month period. There were some outages in our collection, due to network disconnection and unforeseen software issues. We identified these using the

		Commit delay (seconds)	
Time interval	# Txs announced	Median	95^{th} percentile
8-13 Jan 2017	220k	195	323
5-10 Feb 2017	234k	192	301
5-10 Mar 2017	281k	195	307
2-7 Apr 2017	433k	199	339

Table IV: Observation intervals for longitudinal comparison

median block time per hour: if the median for a given hour m_h deviated from the overall median over the entire data set m_o by more than one standard deviation σ, we treated that hour as an outlier—(*i.e.,* $m_h > m_o + \sigma$ or $m_h < m_o - \sigma$). For such outlier periods, we disregarded the directly affected hours as well as one hour prior and subsequent to these hours.

The resulting filtered dataset spans a total duration of 97.38 days, during which we observed 621,865 *blocks*, including uncle blocks. Those blocks contained 6,152,030 transactions, out of which 5,696,471 were unique. 455,559 transaction in the blocks (7.4%) were duplicates. The number of unique transactions that were *announced* to our observation node during the experiment was 5,885,603. Interestingly, this was not a superset of the transactions included in blocks: we did not register an announcement for 24,765 of the unique, included transactions. Moreover, for 21,330 transactions, we received the blocks containing them prior to the announcement message for the transactions. We did not consider such transaction announcements in any of the analyses in this paper. Out of the 621,865 blocks, 34,044 were later referenced as uncle blocks on the main chain and 1731 were 'unrecognized' uncle blocks (not referenced as uncles on the main chain). In addition, we observed 99 forked chains of length 2, and a single forked chain of length 3. For a longitudinal comparison, we extracted 4 *time intervals* of 5 days each, for which we had consecutive uninterrupted data. Table IV lists the details. The interval start dates differ by 4 weeks exactly, and all times are midnight UTC.

C. From First Inclusion to Commit

As Ethereum's transaction handling and inter-block time differ significantly from Bitcoin, there is always a chance of a chain fork. Ethereum recommends to wait for 11 confirmations after block inclusion before assuming that a transaction is committed permanently with high probability. At the time a fork occurs, there is usually no certainty as to which branch will be permanently kept in the blockchain and which branches will be discarded. In particular, transactions that were only included in uncles need to go back to the transaction pool. Before investigating the factors that cause commit delays, it is therefore an interesting question how fast transactions proceed from first inclusion to commit, and how many transactions that were already included in one block actually do get lost as a result of a different branch becoming the main branch. If this fraction is extremely small, this may allow the development of applications that are tolerant to rare transaction losses while profiting from having to wait for fewer confirmation blocks.

Fig. 5 depicts the distribution of the time it takes for an Ethereum transaction to be included in a block and a number of subsequent confirmation blocks. We measured as follows. When our observation node received the announcement of a

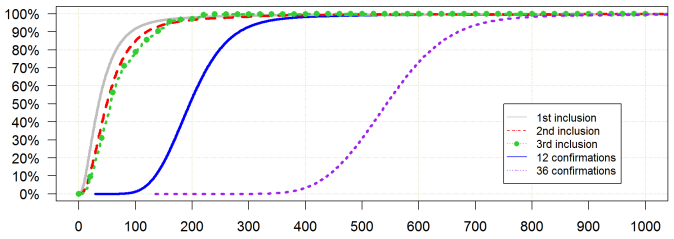

Figure 5: Time (sec) for first inclusion and commit (12 or 36 confirmations), as well as second and third inclusion of transactions that were previously not included in main chain.

block B_1, we analysed the transactions that are included in that block to determine which of the so far *announced* transactions were included. If a transaction Tx_1 was previously announced and was now included, we took the difference between the timestamps recorded for the announcement of Tx_1 and for B_1 as the delay for the first inclusion. In case B_1 became an uncle, all the transactions in B_1 would need to be included in the subsequent blocks. When Tx_1 was included in some subsequent block, say B_2, we used the timestamp of B_2 to calculate the delay of the second inclusion, third inclusion etc. Deeper forks of the chains were treated analogously. The confirmation delay with 12 and 36 blocks for Tx_1 was then calculated from the last block in which Tx_1 was included.

As shown in Fig. 5, the inclusion times tend to follow similar curves. However, compare the slopes of the curves for first to third inclusion to the slopes for twelfth and 36th inclusion: the latter are less steep, indicating the growing fraction of transactions that have to wait longer for a 'commit'. For a '12-block commit', the median waiting time is around 200 seconds, and even the third quantile is not much higher. But the more blocks we require for a commit (say, 24 or 36 blocks), the more likely it becomes that a transaction needs (even considerably) longer than the median would suggest.

Concerning transactions that become 'unincluded', however, we find that these are rare indeed. We observed that 113,122 first transaction inclusions (0.021%) were not permanent; and the same is true for 2602 second inclusions (0.0005%), and 41 of the third inclusions (0.000007%).

D. Impact of User-defined Gas Price and Limit

Ethereum has two user-defined parameters around the concept of gas, namely the gas price and the maximum gas offered for including a given transaction. We proceeded to investigate how these affect the commit times. In particular, we were interested to see if it is possible to speed up the commit time by offering particularly high rewards for miners and, *e.g.,* setting a high gas price.

Based on our collected data, we analysed the effect of the user-defined gas price on the time it took for the transaction to be committed. The CDFs in Fig. 6 depict this relation for five bands of gas price (all in Gwei[7]): $[0, 0], (0, 20), [20, 25), [25, 105), [105, +\infty)$. We chose these five bands for the following reasons. 0 by itself is a special case. The default gas price is 20 Gwei, and the market rate in

[7] 1 Ether are 10^{18} wei.

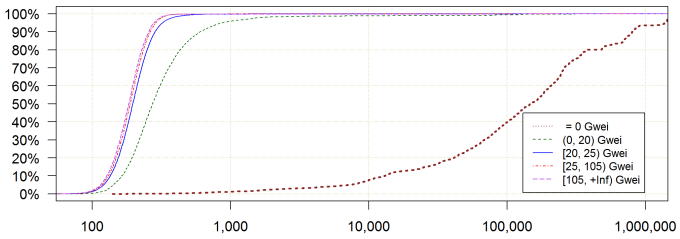

Figure 6: Commit delay (sec) for transactions based on gas price. Note logarithmic x axis.

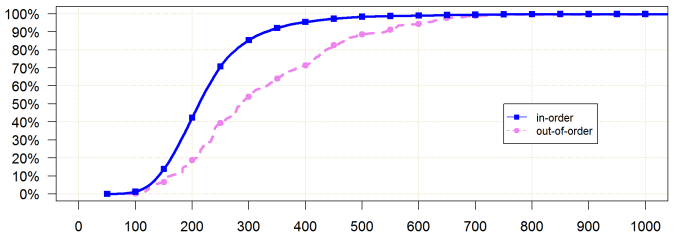

Figure 7: Commit delay (sec) for transactions based on ordering. Note logarithmic x axis.

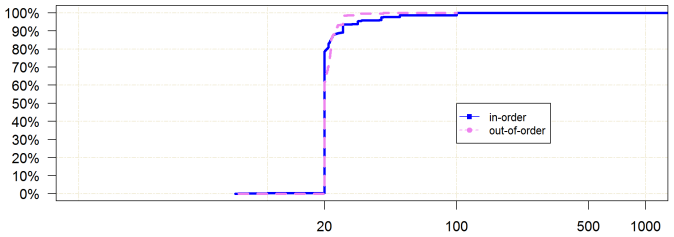

Figure 8: Gas price distribution (GWei) for transactions based on ordering. Note logarithmic x axis.

the observation period was typically between 22 and 24 Gwei[8], so the interval $[20, 25)$ captures 85.57% of all transactions. A cohort of transactions offered around 100 Gwei, and to include these, the next interval was set to $[25, 105)$ Gwei. The final interval covers all remaining gas prices.

As shown in the graph, the higher the gas price in a given band, the less likely we observed long delays. However, we did not observe any meaningful differences from 25 Gwei onwards. Finally, there is a sharp contrast between the 0-band and all other bands: the 0-band has significantly longer commit times.

A second user-defined variable around transaction fees is *maximum gas*, *i.e.,* how much gas execution of the transaction may cost. We analysed its impact on commit delay. While we discovered individual transactions that were delayed due to an exceedingly high gas limit, our analysis was inconclusive: we could not find a strong correlation in any direction between maximum gas and commit delay. This remains an open question for now and warrants longer observation.

E. Impact of network delays

We were also curious whether the Ethereum network suffered from transaction reordering as we had observed it for Bitcoin. Ethereum does not link transactions in the way Bitcoin does, but every transaction has a sequence number ('nonce') which is different for each sender's account. This sequence numbers starts from 0 and increments by 1 for each transaction sent from the same account. It is intended to provide an assurance that transactions from the same account will be executed in a particular deterministic order. However, it also means that a transaction with a nonce $n + 1$ cannot be included into the blockchain unless there is an already included transaction with nonce n—it is 'orphaned'. The transaction with the higher nonce will wait in the transaction pool until the arrival of a transaction with n as nonce.

We hence carried out an experiment that is similar in nature to our previous Bitcoin experiment. We analysed the commit times for *in-order* and *out-of-order* arrival of transactions during the same interval as for our second Bitcoin experiment: 2017-04-13 13:11 UTC–2017-04-14 14:15 UTC. The total number of transaction announcements, which were also committed during this period, was 87,384. The number of transactions with out-of-order nonces was 5403 (6.18%). The commit time for both categories is shown in Fig. 7. The graph suggests that the commit delay for *out-of-order* transactions is almost doubled, compared to *in-order* transactions. To exclude the gas price as a confounding factor, we plot the gas price distribution for both categories, shown in Fig. 8. We did not find a significant difference in gas prices between two categories.

As with Bitcoin, it is hard to rule out other confounding factors that we cannot control for, *e.g.,* Internet routing or overall network connectivity. However, our data allowed us some partial insight into the latter. We inspected transactions with nonce n that were announced *after* transactions with nonce $n + 1$, and compared this with *in-order* transactions announcements. Fig. 9 plots the distribution of unique Ethereum nodes that we saw broadcasting the transaction before inclusion in the block. We find that delayed transactions were known to much fewer nodes. While not conclusive, this provides first indications that network connectivity may have negatively impacted transaction propagation.

V. Impact of the Block Gas Limit in Ethereum

Ethereum has a second form of limit, the so-called *gas limit per block*. Unlike the gas price in a transaction, it is defined by the network of miners and applies to the *sum* of gas consumed by all transactions in a block. If the limit is lower than the gas required for a given transaction, the transaction cannot possibly be included. The development of the gas limit over time is readily available, *e.g.,* on Etherscan[9].

The rationale for the limit is to prevent Denial-of-Service (DoS) attacks on the network by limiting the amount of computation that can be done per bloc. Due to several DDoS attacks to the network, a majority of miners on Ethereum agreed to lower the limit to *approx.* 500,000 gas temporarily – from 15th to 17th of October 2016 according to Etherscan. The network still kept a low limit prior to and after these three days: from 23rd of September 2016 to 22nd of November 2016, with one day exception, the limit was around 2M gas. Around 5 December, it returned to 4M gas. This limitation can negatively impact the inclusion of transactions containing

[8] https://etherscan.io/chart/gasprice – last accessed 13-04-2017

[9] https://etherscan.io/chart/gaslimit

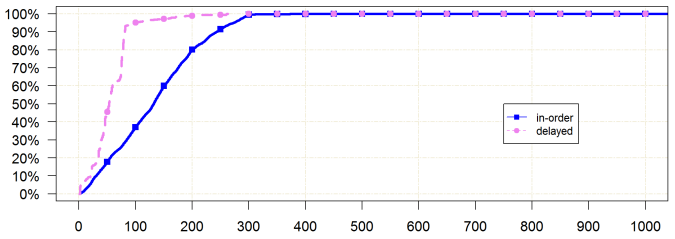

Figure 9: Number of in-order and out-of-order arriving transactions from different peers.

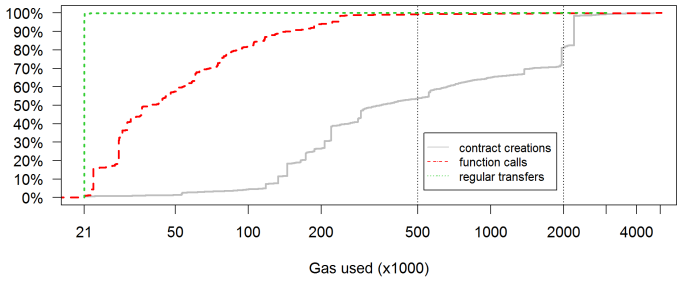

Figure 10: Distribution of gas usage for different types of transactions, prior to DDoS attacks. Dotted vertical lines show limits in response to the attacks.

contracts with much gas. This is not a hypothetical case: in earlier work, we deployed contracts using around 1.5M gas ourselves [3]. However, simple transfer of assets should not be negatively impacted.

We hence chose to investigate whether we could find evidence for this hypothesis in our data. We analysed all transactions that happened before the DoS attacks and used block 2,303,121 as the pre-DoS cut-off block. We considered the amount of gas used for three different types of transactions: financial transfers, regular function calls to contracts, and contract creation.

Figure 10 shows the distribution of gas used for these transaction types. It highlights the gas limits mentioned above as vertical lines. No *financial transfer* transaction used more than 100,000 gas. This was an expected finding, as a financial transfer will incur 21,000 gas as base cost for any transaction, plus possibly a small amount for attached data: between 4 and 68 gas per byte (used *e.g.,* for a description of the transfer, see [9]. As for *function call* transactions, 94% of them used at most 200,000 gas. Only 0.62% of the remaining function call transactions would not have been possible with the 500,000 gas limit. This contradicted a part of our hypothesis and highlighted that most of the functions that are currently in use are not computationally intensive.

However, when inspecting contract creation, we found that only 53.79% of all the contracts created before the DDoS attack could have been created with the 500,000 gas limit, while 46.21% required more gas. This confirmed our hypothesis that many contracts would not have been deployable while the block gas limit was in place. Even for the 2-month period where the network kept the block gas limit at about 2M, 18.78% of contract creation transactions would have been impossible.

VI. Transaction Abort in Ethereum

In this section, we propose a mechanism to artificially abort Ethereum transaction by superseding them with an idempotent or counteracting transaction. This abort mechanism can be useful if, for instance, the system observes that the transaction has not been committed within a specified time frame (as can be the case with, *e.g.,* orphans). As such, the abort mechanism could be implemented to increase the user-friendliness of software clients or wallets.

There are some options to achieve an effect that is similar to an explicit abort. In Ethereum, for instance, the system or user can issue a competing transaction from the same source account, *i.e.,* another transaction with the same nonce. Assume user Alice transfers 1 Ether to Bob by issuing transaction Tx_i with nonce i. After an acceptable time frame, *e.g.,* 10 minutes, has elapsed and Tx_i has not been committed, Alice wants to abort Tx_i. She then submits a new transaction Tx_i', with the same nonce i as specified in Tx_i and a higher transaction fee in order to increase the chances for Tx_i' to be included. For this transaction Tx_i', she does not want to spend more Ether than necessary; thus, she sets the transaction value to 0, and her own account as receiver. Once Tx_i' is committed, Tx_i is superseded by it and becomes outdated. If, in the meantime, Tx_i were to succeed, Tx_i' becomes outdated. This is acceptable, since that was the original intent.

Alternatively to aborting, Alice can 'retry' Tx_i by submitting Tx_i'' as follows: the fields in Tx_i'' contain the same data as in Tx_i, including nonce i—except Alice offers a higher fee for it. Therefore, the hash and digital signature of Tx_i'' will be different from Tx_i, and thus it will be perceived by the miners as a separate transaction. If Alice tried resending Tx_i without any changes, hash and signature would be the same and the miners would not consider it any differently—unless they have previously dropped Tx_i. In the latter case, the reasons for dropping Tx_i might not have changed, and thus the same would likely happen again. If either Tx_i or Tx_i'' succeeds, the respective other transaction would become outdated and invalid, since they both have the same nonce i.

Experiment: Abort Transaction in Ethereum. We tested the above method for abort on the public Ethereum blockchain for three scenarios: 1) a transaction does not get included in the usual period of time; 2) a client changes its mind and decides to roll-back the issued transaction; and 3) a transaction is in indefinite pending state due to insufficient funds. We describe these below in more detail. For all three scenarios, we developed JavaScript implementations with respective parameter settings and timeouts, which we ran on nodejs v4.2.6, using the web3 library v0.17.0-alpha to interact with the geth v1.5.4-stable client. The geth client runs a full blockchain node with mining disabled and interacts with the public network by sending and receiving transactions and newly mined blocks. It only broadcasts newly mined blocks to the rest of the network. We used 0xd8c96ee029945fe1b4272035b704dc52ebcdf051 as the sender account address for all three scenarios, and the results of included transactions can be publicly observed, *e.g.,* on Etherscan[10].

Abort Experiment 1: In order to test the situation where a sent transaction does not get included in the usual timeframe,

[10] https://etherscan.io/address/0xd8c96ee029945fe1b4272035b704dc52ebcdf051

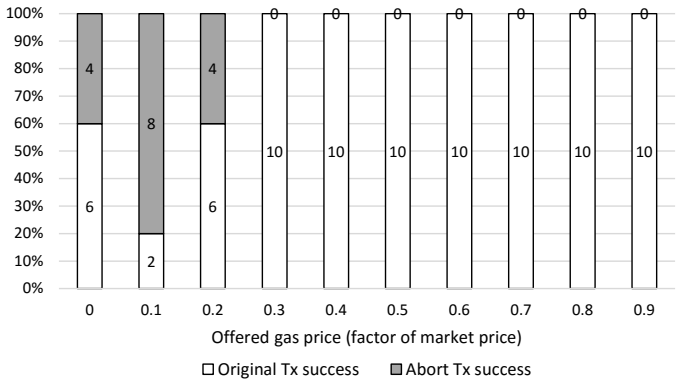

Figure 11: Underbidding market fee and automatic abort after 10 minutes if the original Tx was not included

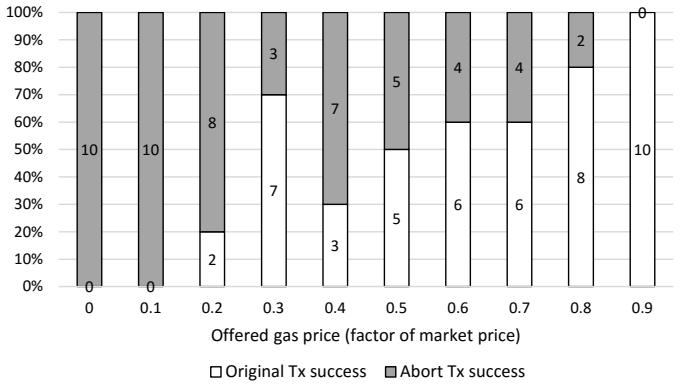

Figure 12: Underbidding market fee and automatic abort after 3 minutes if the original Tx was not included

we submitted 100 transactions that *underbid* the market rate. Specifically, we assumed the average gas price from the previous day (1/12/2016) as market rate (mr), and submitted 10 transactions each for different prices, which are $0, 0.1 \times mr$, $0.2 \times mr$, ..., $0.9 \times mr$. As cut-off time, we rounded up the 99% percentile from our earlier experiment (cf. Fig. 1) to 10 minutes. If the transaction had not been included then, we submitted an abort transaction Tx_{abort} as described above, with the same nonce but at full market rate mr, target 0×0, and value of 0.

The results are shown in Fig. 11. Surprisingly, most transactions were accepted by the network. 6 out of 10 transactions with either 0 or $0.2\times mr$ were accepted. In addition, only 2 out of 10 transactions with $0.1 \times mr$ were accepted. All of the 16 timed-out transactions were successfully aborted with our Tx_{abort} mechanism described above.

Abort Experiment 2: for this experiment, we assumed a client that underbids the market fee and changes its mind regarding an issued transaction. As in the previous experiment, we sent 100 transactions with gas prices 0, $0.1 \times mr$, $0.2 \times mr$, ..., $0.9 \times mr$ for 10 transactions each. Rather than waiting for 10 minutes, we set the timeout value to the target median for Ethereum transaction commit, *i.e.,* 3 minutes.

The results of this experiment are shown in Fig. 12. A much higher percentage of transactions were not included in a block after 3 minutes, in comparison to Fig. 11 with 10 minutes timeout. As before, 100% of Tx_{abort} succeeded. Interestingly, all of them were included in a block after 3 minutes. In two out of the 100 cases, the 3-minute timeout for the original transaction was reached, Tx_{abort} was sent, but the original transaction Tx_{orig} still won the race and got included and committed in the blockchain. Thereby, Tx_{abort} was outdated. As stated above, this is a possibility that clients should be prepared for. The reasons for such a situation include (i) processing time in our client when preparing the Tx_{abort}; (ii) broadcast delays or other network effects where the winning miner does not receive Tx_{abort} before including Tx_{orig}; or (iii) non-rational scheduling of transactions in the pool, where no preference is given to the transaction with the higher fee.

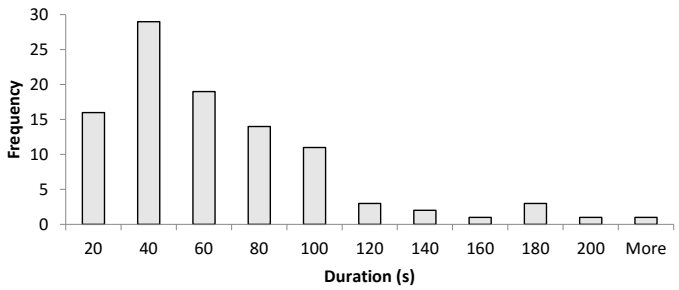

Figure 13: Abort duration histogram, from experiment 3

Abort Experiment 3: in this last experiment we submitted two transactions, creating a situation that corresponds to faulty inputs from a user (or user's program). We have observed such behaviour during our live observation of public Ethereum. To replicate it, we submitted two transactions, Tx_1 and Tx_2, as follows. Assume that the last nonce for the sender address was n and its account balance k. Then we create Tx_1 with nonce $n+1$ and value $\frac{1}{1000}k$ and Tx_2 with nonce $n+2$ and value $\frac{999}{1000}k$. For both transactions, we set the gas price to $0.7 \times mr$. Due to the nonce, Tx_1 must be included before Tx_2. However, due to the positive gas price, the account balance resulting from the inclusion of $Tx_(1)$ is insufficient for Tx_2. Finally, we submit Tx_2, wait 5 seconds, and then submit Tx_1. This gives Tx_2 the chance to get broadcast before Tx_1 is known to any node, including our own. This procedure is needed so that the client submits Tx_2 to the network; since geth is not aware of Tx_1 and its contents when we submit Tx_2, it broadcasts Tx_2. Otherwise, it might detect the insufficient balance and not accept Tx_2.

Once Tx_1 has been included in a block, Tx_2 is invalid due to insufficient funds. However, this does not always get checked, and hence Tx_2 may remain in the transaction pool for a long time. In fact, if another transaction deposited funds into the sender account, Tx_2 would become valid and be executed. This, again, is behaviour that we observed. Here, we send a Tx_{abort} with the nonce $n+2$, to abort Tx_2.

We ran this experiment until we had submitted Tx_{abort} 100 times. All 100 submitted Tx_{abort} were successful. We measured the time it took for Tx_{abort} to be included in a block (first inclusion), and plotted that as shown in Fig. 13. The median for those times is 45 sec, and the maximum 230 sec.

VII. Related Work

The dependability of blockchain systems have been investigated on various aspects. However, most investigations were published as blog posts, in many cases without a sound

description of the methodology. Vitalik Buterin, the creator of Ethereum, wrote a post[11] where he related the *block time*, *i.e.,* the expected interval time between blocks, to security. This post explains that to tolerate Byzantine faults, a faster block time translates into a finer granularity that allows to converge quicker than a longer block time. In a different post[12], he wrote about the rate of uncle blocks in Ethereum. In contrast to our experiments and analyses, this post is written from the viewpoint of the network, whereas we focus on the impact of network effects like uncles on individual transactions and the resulting availability of blockchain-based systems.

When building a distributed system, one has to choose between making it available or consistent. This is a direct implication of the well-known CAP theorem, initially mentioned by Brewer [10] and proved by Gilbert and Lynch [11]. The CAP theorem states the impossibility for a distributed service to provide *(i)* consistency: returning the right response to a request; *(ii)* availability: returning a response to each request; and *(iii)* partition-tolerance: supporting message delays and losses.

Our observations that mainstream blockchains offer limited availability is surprising, given that they usually do not guarantee consistency. Other approaches would typically favour consistency over availability [12], [13]. In contrast, Ethereum transactions are not guaranteed to remain in the order they were committed [8]. It appears that these blockchains typically try to ensure consistency with some probabilities that depend on environmental assumptions, such as the delays of message and the mining power [14]. While recent results showed experimentally that network delays could impact Ethereum consistency [15], we are not aware of any in-depth evaluation of its availability property.

In terms of peer-reviewed academic publications, several works [16], [17] studied the possibility to increase the pace at which blocks could be appended to the chain by benefiting from a leader. An interesting improvement discussed by Sompolinsky and Zohar [16] was the reduced expected interval time between blocks from 10 minutes in Bitcoin to 12–15 seconds in Ethereum, by accounting for "uncle blocks" rather than considering only the longest branch of the chain. Eyal *et al.* [17] proposed to reduce the expected interval time between blocks by electing a leader that can append micro-blocks frequently. This does not necessarily reduce the delay of reaching consensus, as nodes that require high confidence will not necessarily reach consensus faster than with Bitcoin [1].

The commit-time of Ethereum transactions was explored previously [8]. It was shown that the time a transaction takes to commit is proportional to the difficulty of the crypto-puzzles of the system. However, this study experiments only in a private chain context, where the difficulty is significantly lower than in the Ethereum public chain. This explains why the commit-times reported in [8] differs substantially from our public chain observations.

[11]https://blog.ethereum.org/2015/09/14/on-slow-and-fast-block-times/ – last accessed 5/12/2016

[12]https://blog.ethereum.org/2016/10/31/uncle-rate-transaction-fee-analysis/ – last accessed 5/12/2016

VIII. Conclusions

We provided a first, detailed analysis of issues that can negatively impact commit times in permissionless proof-of-work blockchains, and a way to limit this effect with the introduction of an explicit abort mechanism.

In particular, we warn application developers against the factors that may dramatically affect their quality of service. We found that network reordering can impact dramatically commit times and even counterbalance the effects of transaction fees and gas price. Furthermore, we could also show that measures taken by the Ethereum community to counter DoS attacks had a strong availability impact on contract creation.

For the future, we plan to study how availability issues can impact the execution of smart contracts as these are becoming more important in the Ethereum blockchain.

References

[1] S. Nakamoto, "Bitcoin: a peer-to-peer electronic cash system," 2008, http://www.bitcoin.org, last accessed 5/12/2016.

[2] M. Walport, "Distributed ledger technology: Beyond blockchain," UK Government Office for Science, report GS/16/1, 2016.

[3] I. Weber, X. Xu, R. Riveret, G. Governatori, A. Ponomarev, and J. Mendling, "Untrusted business process monitoring and execution using blockchain," in *Intl. Conf. on BPM*. Springer, Sep. 2016.

[4] X. Xu, C. Pautasso, L. Zhu, V. Gramoli, A. Ponomarev, A. B. Tran, and S. Chen, "The blockchain as a software connector," in *WICSA2016*, Venice, Italy, Apr. 2016.

[5] V. Gramoli, "On the danger of private blockchains," in *Workshop on Distributed Cryptocurrencies and Consensus Ledgers (DCCL'16)*, 2016.

[6] I. Eyal and E. G. Sirer, "Majority is not enough: Bitcoin mining is vulnerable," in *International Conference on Financial Cryptography and Data Security*. Springer, 2014, pp. 436–454.

[7] C. Decker and R. Wattenhofer, "Information propagation in the Bitcoin network," in *Proc. IEEE Conf. peer-to-peer networks (P2P)*, 2013.

[8] C. Natoli and V. Gramoli, "The blockchain anomaly," in *Proceedings of the 15th IEEE International Symposium on Network Computing and Applications (NCA'16)*, Oct 2016.

[9] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," 2015, yellow paper.

[10] E. Brewer, "Towards robust distributed systems," in *Proc. 19th ACM Symposium on Principles of Distributed Computing (PODC)*, July 2000.

[11] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *ACM SIGACT News*, vol. 33, pp. 51–59, 2002.

[12] C. Cachin, "Blockchain - from the anarchy of cryptocurrencies to the enterprise," in *Proc. 20th Int'l Conference on Principles of Distributed Systems (OPODIS)*, 2016, keynote presentation.

[13] T. Crain, V. Gramoli, M. Larrea, and M. Raynal, "(leader/randomization/signature)-free byzantine consensus for consortium blockchains," arXiv, Tech. Rep. 1702.03068, 2017.

[14] C. Natoli and V. Gramoli, "The balance attack against proof-of-work blockchains: The R3 testbed as an example," arXiv, Tech. Rep. 1612.09426, 2016.

[15] ——, "The balance attack or why forkable blockchains are ill-suited for consortium," in *The 47th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'17)*. IEEE, Jun 2017.

[16] Y. Sompolinsky and A. Zohar, "Secure high-rate transaction processing in bitcoin," in *Intl. Conf. Financial Cryptography and Data Security FC*, 2015, pp. 507–527.

[17] I. Eyal, A. E. Gencer, E. Sirer, and R. van Renesse, "Bitcoin-NG: A scalable blockchain protocol," in *USENIX NSDI*, 2016.