BOSTON UNIVERSITY

COLLEGE OF ENGINEERING

Dissertation

MEASURING AND IMPROVING THE PERFORMANCE OF THE BITCOIN NETWORK

by

MUHAMMAD ANAS IMTIAZ

B.Sc., National University of Computer & Emerging Sciences, 2014
M.S., Boston University, 2022

Submitted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

2022

© 2022 by
MUHAMMAD ANAS IMTIAZ
All rights reserved

Approved by

First Reader

David Starobinski, Ph.D.
Professor of Electrical and Computer Engineering
Professor of Systems Engineering

Second Reader

Ari Trachtenberg, Ph.D.
Professor of Electrical and Computer Engineering
Professor of Systems Engineering

Third Reader

Gianluca Stringhini, Ph.D.
Assistant Professor of Electrical and Computer Engineering

Fourth Reader

Alan Liu, Ph.D.
Assistant Professor of Electrical and Computer Engineering
Assistant Professor of Computer Science

"The problem, you see, when you ask why *something happens, how does a person answer why something happens? . . . when you explain a* why, *you have to be in some framework that you allow something to be true. Otherwise, you're perpetually asking why. . .. If you try to follow anything up, you go deeper and deeper in various directions. . .. I'm telling you how difficult the* why *question is. You have to know what it is that you're permitted to understand and allow to be understood and known, and what it is you're not."*

RICHARD FEYNMAN *on* Magnets (1983)

Acknowledgments

Alhamdulillah! What amazing four-something years these have been! I have learnt a great deal – both academic and otherwise – during my post-graduate journey at Boston University and I hope my work will make positive impact(s) in the world of computer engineering. I am grateful to the Department of Electrical and Computer Engineering for allowing to me the opportunity and providing the necessary support to pursue a doctorate in computer engineering. I am thankful to my dissertation committee, comprising of Professors David Starobinski, Ari Trachtenberg, Gianluca Stringhini, and Alan Liu, for their valuable feedback in improving this thesis.

In his words, Prof. Starobinski hired me to do research on automotive systems given my background in software development of automotive network stacks at Mentor Graphics. However, that is not what this dissertation turned out to be about... Instead, Prof. Starobinski allowed me the freedom to pick a project of my own choice from the ones that were going on at the time at NISLab. Fast-forward a few years and here we are with some of our remarkable work in the field of blockchains! I will always appreciate the role that Prof. Starobinski played as my advisor and I am thankful to him for guiding me throughout my Ph.D. career. I have found nothing but support and constructive feedback from him. There has not been a single moment where I was afraid of speaking with him about things that were not working because I was always confident that he would hear me out, and give some sound advice. I would also like to acknowledge Prof. Trachtenberg for always providing to-the-point feedback and comments that have helped improve my paper writing and presentation skills over the years.

I am indebted to every professor and teacher who has ever taught me anything. They have not only played an important part in developing my technical skills but have also helped mold my personality. The list is very long and I cannot possibly mention everyone though I would specifically like to acknowledge Profs. Starobinski, Egele, Coskun, Moreshet, Varia, Trachtenberg, and Liu from my time at Boston University; Aftab Alam, Irfan Iqbal, Omer Saleem Bhatti, Azeem Hafeez, Shazia Haque, Zahra Arshad, and Nabeel A. Qadeer from National University of Computer & Emerging Sciences; and Mussarat Mashadi, and Zakia Sarwar.

I have had the privilege of having some great lab mates during my time at NISLab: Johannes, Nabeel, Steven, Trishita, Liangxiao, Stefan, Jonathan, Novak, Zhenpeng, Bowen, Daniel, John, and Sean. We have had many fun debates in the lab and on Slack. I have learnt many new things from all of them be it in our random discussions or research/course collaborations. I have had some really amazing friends in my life, some of whom have gone above and beyond to look out for me: Shehwar, Salman, Natasa, Waleed A., Usman, Kasim, Zaki, Gulrayz, Waleed T., Ahmad, Golsana, Umair, Jawad, Ali, Imran, Salahuddin, Asjad, Burhan, and Akbar. *I hope our friendship will outlast our fame!*

I dedicate this dissertation to my late father whom I miss dearly and wish were with me to celebrate this achievement, to my mother who ensured quality education for her children despite facing several hardships, to my brother who never complained when he had to take on many of my responsibilities when I left home for U.S. to start my Ph.D., to my sisters who always supported and encouraged me to pursue my dreams, and to my beautiful wife for making my life so much better with her presence, and unconditional love and affection. I am also grateful to my extended family, especially my maternal grandfather, and to my in-laws for their care and never-ending support.

Finally, I thank everybody who played a part in the successful realization of this thesis, whose name I could not mention above.

MEASURING AND IMPROVING THE PERFORMANCE OF THE BITCOIN NETWORK

MUHAMMAD ANAS IMTIAZ

Boston University, College of Engineering, 2022

Major Professor: David Starobinski, Ph.D.
Professor of Electrical and Computer Engineering
Professor of Systems Engineering

ABSTRACT

The blockchain technology promises innovation by moving away from conventional centralized architectures, where trust is placed in a small number of actors, to a decentralized environment where a collection of actors must work together to maintain consensus in the overall system. Blockchain offers security and pseudo-anonymity to its adopters, through the use of various cryptographic methods. While much attention has focused on creating new applications that make use of this technology, equal importance must be given to studying naturally occurring phenomena in existing blockchain ecosystems and mitigating their effects where harmful.

In this dissertation, we develop a novel open-source *log-to-file* system that provides the ability to record information relevant to events as they take place in *live* blockchain networks. Specifically, our open-source software facilitates *in-situ* measurements on full nodes in the live Bitcoin and Bitcoin Cash blockchain networks. This measurement framework sheds new light on many phenomena that were previously unknown or scarcely studied.

First, we examine the presence and impact of churn, namely nodes joining and leaving, on the behavior of the Bitcoin network. Our data analysis over a two-month period shows that a large number of Bitcoin nodes churn *at least* once. We perform statistical distribution fitting to this churn and emulate it in our measurement nodes to evaluate the impact of churn on the performance of the Bitcoin protocol. From our experiments, we find that blocks received by churning nodes experience as much as five times larger propagation delay than those received by non-churning nodes. We introduce and evaluate a *novel* synchronization scheme to mitigate such effects on the performance of the protocol. Our empirical evaluation shows that blocks received by churning nodes that synchronize their mempools with peers have roughly *half* the delay in propagation experienced by those that do not synchronize their mempools.

We next evaluate and compare the performance of three block relay protocols, namely the *default* protocol, and the more recent *compact block* and *Graphene* protocols. This evaluation is conducted over *full nodes* running the Bitcoin Unlimited client (which is used in conjunction with the Bitcoin Cash network). We find that in most scenarios, the Graphene block relay protocol outperforms the other two in terms of the block propagation delay and the amount of total communication associated with block relay. An exception is when nodes churn frequently and spend a significant fraction of time off the network, in which case the compact block relay protocol performs best. In-depth analyses reveal subtle inefficiencies of the protocols. Thus, in the case of frequent churns, the Graphene block relay protocol performs as many as *two* extra round-trips of communication to recover information necessary to reconstruct blocks. Likewise, an inspection of the compact block relay protocol indicates that the full transactions included in the initial block message are either unnecessary or insufficient for the successful reconstruction of blocks.

Finally, we investigate the occurrence of orphan transactions which 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. Our data reveals that slightly less than half 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, a larger size, and a lower transaction fee per byte than all other received transactions. Moreover, the network overhead incurred by these orphan transactions can be significant when using the default orphan memory pool size (*i.e.,* 100 transactions), although this overhead can be made negligible if the pool size is simply increased to 1,000 transactions.

In summary, this dissertation demonstrates the importance of characterizing the inner behavior of the peer-to-peer network underlying a blockchain. While our results primarily focus on the Bitcoin network and its variants, this work provides foundations that should prove useful for studying and characterizing other blockchains.

Contents

Contents **x**

List of Tables **xv**

List of Figures **xvii**

1 Introduction **1**

- Research questions . . . 5
- Contributions . . . 6
- Takeaways . . . 13
- Road map . . . 16

2 Background and related work **17**

- 2.1 Preliminaries . . . 17
  - 2.1.1 Transaction . . . 17
  - 2.1.2 Block . . . 18
  - 2.1.3 Miner . . . 20
  - 2.1.4 Blockchain . . . 20
- 2.2 Bitcoin and Bitcoin Unlimited . . . 21
  - 2.2.1 Bitcoin . . . 22
  - 2.2.2 Bitcoin Unlimited . . . 23
  - 2.2.3 Data structures . . . 24
    - Bloom filter . . . 24
    - Invertible Bloom lookup table . . . 26

2.2.4 Block relay protocols 27
Default (normal) block relay 27
Compact block relay 28
Graphene block relay 30
2.3 Orphan transactions 34
2.4 Related work 36
2.4.1 Measurement tools 36
2.4.2 Block propagation and churn 37
2.4.3 Orphan transactions 41
2.4.4 Summary 42

3 Churn in the Bitcoin network 44

3.1 Churn characterization 45
3.1.1 Obtaining and processing data 45
3.1.2 Churn rate 47
3.1.3 Statistical fitting of session lengths 48
Up sessions 49
Down sessions 51
3.1.4 Subnet analysis 52
3.1.5 Geographic analyses 54
3.2 Experimental analysis of compact block performance with churn . . . 57
3.2.1 Data collection mechanism 57
3.2.2 Experimental setup 58
3.2.3 Statistics on compact blocks 59
3.2.4 Statistics on missing transactions 60
3.2.5 Statistics on propagation delay 61
3.3 MempoolSync 62

3.3.1 Design of MempoolSync 63
3.3.2 Experimental evaluation of MempoolSync in the presence of churn 68
3.3.3 Statistics on compact blocks 71
3.3.4 Statistics on missing transactions 71
3.3.5 Statistics on propagation delay 72
3.4 Discussions and limitations 73
Characterization of churn 73
Sampled session lengths 75
MempoolSync 75
3.5 Summary 76

4 Empirical comparison of block relay protocols for blockchains 78

4.1 Evaluation of block relay protocols 79
4.1.1 Data collection mechanism 79
4.1.2 Experimental setup 80
4.1.3 Statistics on the propagation delay of blocks 82
4.1.4 Statistics on the communication size per block 85
4.1.5 Correlation between propagation delay and communication per block 88
4.2 Insights into block relay protocols 90
4.2.1 Graphene in depth 90
4.2.2 Temporal analysis of the Graphene block relay protocol 93
4.2.3 Size of first message across block relay protocols 96
4.2.4 On the usefulness of additional transactions in the compact block relay protocol 98
4.3 Summary 102

5 Orphan Transactions in the Bitcoin Network 104

5.1 Characterization of orphan transactions 105
  5.1.1 Measurement setup 106
  5.1.2 Number of parents 106
  5.1.3 Transaction fee of missing parents 108
  5.1.4 Transaction size of missing parents 109
  5.1.5 Relating transaction fee to size of missing parents 110
  5.1.6 Orphan transactions in blocks 112
  5.1.7 Delay in receiving missing parents from peers 113
  5.1.8 Impact of transaction fee 115
5.2 Comparison of orphan transaction behavior with different orphan pool parameters 117
  5.2.1 Measurement setup 117
  5.2.2 Removal of orphan transactions from orphan pool 120
  5.2.3 Addition of orphan transactions to orphan pool 122
  5.2.4 Network overhead 123
  5.2.5 Performance overhead 124
    CPU overhead 124
    Memory overhead 125
  5.2.6 Varying orphan transaction timeouts 127
5.3 Orphan transactions in nodes joining the network 128
  5.3.1 Measurement setup 129
  5.3.2 Fraction of orphan transactions 129
  5.3.3 Arrival times of orphan transactions 130
  5.3.4 Removal of orphan transactions from orphan pool 131
5.4 Discussions and limitations 132

Peer selection in measurement nodes 134

Performance impact of orphan transactions 134

Ideas for future development 134

5.5 Summary 135

6 Conclusions and future work 137

Summary of contributions and findings 138

Future work directions 142

Concluding remarks 144

A Explanation of log-to-file system 145

A.1 Motivation and challenges 145

A.2 Design of the system 148

A.3 Usage of the system 148

Bibliography 155

Curriculum Vitae 179

List of Tables

3.1 R^2 and RMSE scores of distribution fits for “up session” lengths. . . . 50

3.2 R^2 and RMSE scores of distribution fits for “down session” lengths. . 52

3.3 Percentage of continuously connected nodes in each continent. 56

3.4 Block reception statistics for control nodes C_1, C_2, C_3, and C_4, and churning nodes X_1, X_2, X_3 and X_4. 60

3.5 An illustration of **(a)** *unsorted* transactions in the mempool with their ancestor scores (in *satoshis)*, **(b)** *sorted* transactions in the mempool with their ancestor scores (in *satoshis)*, **(c)** hashes of transactions already sent to a peer, and **(d)** transaction hashes sent in `MempoolSync` message when $N = 5$. 69

3.6 Block reception statistics for churning nodes M_1, M_2, M_3, and M_4 that accept `MempoolSync` messages, and churning nodes X_1, X_2, X_3, and X_4 that do not accept such messages. 71

4.1 Fraction of blocks that have propagation delay larger than 100 ms, and 1,000 ms in Graphene, compact and default block relay protocols over varying fluctuating periods with **(a)** 25%, and **(b)** 75% off duty cycles. 84

4.2 Fraction of blocks that have communication sizes larger than 10 kB, 100 kB, and 1,000 kB in Graphene, compact and default block relay protocols over varying fluctuating periods with **(a)** 25%, and **(b)** 75% off duty cycles. 87

4.3 Coefficients for Spearman Rank Correlation between the block propagation delays and block communication sizes in Graphene, compact, and default block relay protocols in nodes in the `statistical churn` regime. 89
4.4 Coefficients for Spearman Rank Correlation between the block propagation delays and block communication sizes in Graphene, compact, and default block relay protocols in nodes in the `periodic churn` regime. In general, the propagation delays and communication size are moderately to highly correlated. 89

5.1 Average CPU usage of nodes with different orphan pool sizes. 126

List of Figures

2·1 A Bitcoin transaction with multiple input accounts and multiple output accounts. The difference between the total input value and the total output value is the transaction fee. 18

2·2 An illustration of a block. 19

2·3 An illustration of a merkle tree created by transactions $\texttt{tx}_\texttt{A}$, $\texttt{tx}_\texttt{B}$, $\texttt{tx}_\texttt{C}$, and $\texttt{tx}_\texttt{D}$. The node with the value $\texttt{H}_\texttt{ABCD}$ is called the *merkle root* of the tree. 19

2·4 An illustration of a blockchain. Each block is cryptographically linked to the previous creating an unalterable chain of blocks. 21

2·5 Illustration of Bloom filter operations. 25

3·1 Size of Bitcoin network over the measurement period. 47

3·2 Daily churn rate on the Bitcoin network. 48

3·3 Distribution fits for “up session” lengths. 50

3·4 Distribution fits for “down session” lengths. 51

3·5 Largest IPv4 /24 subnets sorted in descending order. 53

3·6 Number of reachable nodes in the largest IPv4 /24 subnet in consecutive Bitcoin network snapshots. 53

3·7 CDF of duty cycle of nodes in the largest IPv4 /24 subnet. The duty cycle of a node represents the fraction of a time it is reachable during the measurement period. 55

3·8 CDF of duty cycle of nodes in the 2^{nd} to 10^{th} largest IPv4 /24 subnets. 55

3·9 Correlation matrix showing the correlation between churn behavior of nodes in the 10 largest /24 IPv4 subnets. The red line delimiters separate between different subnets. 56

3·10 Geographic location of individual nodes on the Bitcoin network. Nodes that are always up are marked white. Remaining (black) nodes contribute to churn in the network. 57

3·11 Sampled up and down session lengths. 58

3·12 CCDF of number of missing transactions in churning and control nodes. 61

3·13 Propagation delay across all blocks for both churning and control nodes. 62

3·14 Exchange of messages between the non-churning node (*sender*) and the churning node (*receiver*) in the `MempoolSync` protocol. 64

3·15 Procedure for selecting transaction hashes to be included in the `inv` message in each round. 65

3·16 Sampled up and down session lengths. 70

3·17 CCDF of number of missing transactions across all blocks for all nodes. 72

3·18 CCDF of propagation delay across all blocks for all nodes. 73

4·1 Complementary cumulative distribution functions (CCDFs) of block propagation delays in Graphene, compact, and default block relay protocols in the `always on` and `statistical churn` regimes. Graphene block relay protocol performs best in roughly 99% of blocks whereas default block relay protocol always performs worst. 83

4·2 CCDFs of block communication sizes in Graphene, compact, and default block relay protocols in the ***`always on`*** and ***`statistical churn`*** regimes. Graphene block relay protocol performs best in both regimes whereas default block relay protocol performs worst. 86

4·3 Proportion of block decode failures, *i.e.,* scenarios ③, ④, and ⑤ in **Listing** 3, over different fluctuation periods with 25% and 75% off duty cycles. Block decode failure rates are higher when nodes churn more often and stay off the network longer thereby not being able to recover. This is prominent in fluctuating periods of 20 m and 1 hr. . . . 91

4·4 Average number of missing transactions (with 95% confidence intervals) from blocks that are decoded successfully, *i.e.,* scenario ② in **Listing** 3 over different fluctuation periods with 25% and 75% off duty cycles. 92

4·5 Average number of missing transactions from blocks that are decoded successfully, *i.e.,* scenario ④ in **Listing** 3 over different fluctuation periods with 25% and 75% off duty cycles. 93

4·6 Percentage of blocks received in the `statistical churn` regime that face the five scenarios after the churning nodes rejoin the network. The longer a node stays on the network, the more scenario ① (i.e., no extra round-trip) prevails whereas the scenarios ③, ④, and ⑤ do not occur very often. 94

4·7 Percentage of blocks received in the `periodic churn` regime with a fluctuation period of 1 hr and off duty cycle of **(a)** 25% and **(b)** 75% that face the five scenarios after a node rejoins the network. In either case, scenario ① does not represent the majority of cases for the first block and scenarios ③ and ⑤ occur infrequently. 95

4·8 Sizes of the first block messages, *i.e.,* `block`, `cmpctblock`, and `grblk`, against the number of transactions in the respective compact blocks. The `block` messages almost always have the largest sizes. 98

4·9 Sizes of the `cmpctblock` messages (left y-axis) and the number of additional transactions in blocks (right y-axis) against the number of transactions in the respective compact blocks. There exists a direct correlation between the number of additional transactions in and the sizes of the `cmpctblock` messages 99

4·10 Number of useful additional transactions in `cmpctblock` messages against the total number of additional transactions in respective `cmpctblock` messages in (lower half) always on and (upper half) statistically churning nodes. The diagonal ($x = y$) represents the case when 100% of the additional transactions in `cmpctblock` messages are useful. 101

5·1 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. 107

5·2 Cumulative distribution functions (CDFs) of transaction fee of missing parents of orphan transactions, and transaction fee of all other transactions. 108

5·3 CCDFs of transaction size of missing parents of orphan transactions, and transaction size of all other transactions. 110

5·4 CDFs of transaction fee per byte of missing parents of orphan transactions, and transaction fee per byte of all other transactions. 111

5·5 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. 112

5·6 CCDF of time elapsing from the point a transaction becomes orphan till one of its parents is found. 114

5·7 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. 115

5·8 CCDFs of transaction fee of parent transactions that are relayed with and parent transactions that are relayed before their respective child transactions. 116

5·9 Similarity matrix depicting average number of common peers across nodes during the first round of measurement period. 119

5·10 Similarity matrix depicting average number of common peers across nodes during the second round of measurement period. 120

5·11 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. 121

5·12 Number of unique and total number of orphan transactions received across nodes with varying orphan pool sizes. 122

5·13 Network overhead incurred by nodes with varying orphan pool sizes across nodes. 125

5·14 Fraction of orphan transactions that are removed from the orphan pool due to each of the six causes across all nodes, under different timeouts. 128

5·15 Percentage of transactions that become orphan during each 5-minute bin interval of the 12 hour long sessions. 130

5·16 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. 131

5·17 CDF of arrival times of orphan transactions during measurement periods. Roughly 50% of all orphan transactions are received in the first two hours. 132

5·18 Fraction of transactions that are removed from the orphan pool due to each of the six causes (see **Section 2.3**) over the first two hours of the 12 hour long sessions. 133

A·1 State transitions for the default block relay protocol from the perspective of a receiver. 146

A·2 State transitions for the compact block relay protocol from the perspective of a receiver. 146

A·3 State transitions for the Graphene block relay protocol from the perspective of a receiver. States with backgrounds in green, purple, yellow, cyan, and red occur in scenarios ①, ②, ③, ④, and ⑤, respectively (see **Listing 3**). A state with multiple background colors represents more than one corresponding scenarios that transition through it. 147

List of Abbreviations

API	Application programming interface
BCH	Bitcoin Cash
BTC	Bitcoin
BU	Bitcoin Unlimited
CCDF	Cumulative distribution function
CDF	Complementary cumulative distribution function
CPI	Characteristic polynomial interpolation
CPU	Central processing unit
EST	Eastern Standard Time
GB	Gigabyte (*i.e.,* 10^9 bytes)
GHz	Gigahertz
HDD	Hard disk drive
IBLT	Invertible Bloom lookup table
ID	Identifier
IoT	Internet of Things
IP	Internet protocol
IPv4	Internet protocol version 4
IPv6	Internet protocol version 6
JSON	JavaScript Object Notation
KB	Kilobytes (*i.e.,* 10^3 bytes)
LTS	Long-term support
MB	Megabyte (*i.e.,* 10^6 bytes)
MLE	Maximum likelihood estimation
NAT	Network address translation
P2P	Peer-to-peer
PCN	Payment channel network
RAM	Random access memory
RBF	Replace-by-Fee
RMSE	Root mean squared error
RPC	Remote procedure call
SRC	Spearman Rank Correlation
TB	Terabyte (*i.e.,* 10^{12} bytes)
TX	Transaction
USD	United States dollar

Chapter 1

Introduction

Blockchain has emerged as a disruptive technology with applications in digital currencies (also known as *cryptocurrencies*) [1–6], supply chain [7–10], health care [11–14], the Internet of Things (IoT) [15–20], and more [21–28]. Blockchains are *immutable* in the sense that it is practically impossible (as long as the underlying cryptography is secure) for an adversary to tamper with the data recorded in a blockchain without controlling a significant amount of resources [1,29]. They are also *decentralized* and reduce the amount of trust placed in a single actor such as central banks in conventional payment methods. Instead, blockchains require an entire network of users to establish and maintain consensus, often with economic incentives [30]. These properties make the blockchain technology suitable for the aforementioned applications, attracting attention from a wide user base [31]. It is estimated that more than 220 million people have used cryptocurrencies *alone* by June 2021 [32] and this number is expected to increase as new blockchain applications are created.

Bitcoin was the first application of blockchain, and was originally introduced by Satoshi Nakamoto in 2008 [1] as a peer-to-peer electronic payment system. It is still one of the largest and most popular blockchain systems to date and is currently used for buying and selling a wide variety of goods in different markets across the globe including Virgin Galactic [33], AT&T [34], Newegg [35], and more. At the time of writing, this cryptocurrency has a total market capitalization of well over *one trillion* USD [36]. Numerous spin-off cryptocurrencies, called *altcoins*, have been launched since the inception of Bitcoin, with a total market cap of more than 1.5 trillion USD [37]. *With such huge amounts of money at stake, blockchains present compelling areas for research.*

Despite possessing impressive properties that make them appealing for several use cases, blockchains may still be vulnerable to attacks. Quick dissemination of *transactions* (*i.e.,* transfers of tokens such as cryptocurrency, digital assets, etc.,) and *blocks* (*i.e.,* collections of transactions; explained in more detail in **Chapter 2**) is vital to maintaining consensus and security in a trust-less blockchain network. Slow propagation of transactions may result in nodes in the network to *miss* transactions causing added extra round-trip communication and, consequently, increasing the delay in propagation of blocks containing these transactions. Similarly, out-of-order propagation of transactions may render them *orphan* [38,39], *i.e.,* a transaction whose *parent* transaction(s) are not known and, thus, the validity of the tokens it spends cannot be verified. This not only hinders relay of such transactions in the network, but orphan transactions have also been used to infer topology of [40] and mount attacks [41,42] on the Bitcoin network.

Additionally, slow propagation of blocks can cause an increase of *forks* of the blockchain, wherein several blocks are mined independently and distributed before the network nodes accept one of the blocks as the head of the blockchain while the other blocks become *stale*[1]. This issue leads to periods of ambiguity, during which different nodes in the network may have different views of the blockchain. An adversary may leverage such ambiguities for certain attacks, such as the double spending, selfish mining, and eclipsing attacks [44–49]. Stale blocks also lead to a waste of computational resources for nodes that have created them, and the nodes that have created new blocks on top of them.

---

[1] The terms *stale* and *orphan* blocks are often interchangeably used in the blockchain community to refer to blocks that are no longer part of the main chain [43]. We use the term *stale* here so as not to confuse the reader between "orphan transactions" and "orphan blocks" which are quite different.

The rate at which transactions are added to blocks determines the number of transactions that a blockchain system can support, and it may thus seem intuitive that increasing block size can lead to higher transaction throughput [50–53]. Unfortunately, large blocks also increase delay in the propagation of blocks as well as require more bandwidth for propagation and may consume more network resources.

To address the challenge of information dissemination, several ideas have been proposed to speed up block propagation in various blockchain networks including block relay protocols such as the *compact* and *Graphene* block relay protocols that compress sizes of blocks [54–57], protocols such as FIBRE that enable high-speed transfer of blocks within a network [58–60], and more [61–67]. However, not all ideas have been implemented or evaluated in a real blockchain system. For instance, some ideas, such as *Kadcast* [67] and *P4P* [64], have only been simulated or have only been evaluated over simulated data sets. These proposals, nevertheless, inherently assume that nodes in the blockchain system are always connected to the network. That is, they do not account for *churn*, the effect created by independent arrival and departure of nodes in a peer-to-peer network.

In general, block relay protocols need to be resilient to churn of *full nodes*, which is ubiquitous and pervasive in blockchain networks [68–70]. Full nodes perform the process of verifying the integrity and correctness of blocks [71, 72]. They each store the entire blockchain [73], while also forming a peer-to-peer network for updates. Full nodes verify that a new mined block meets specifications and is valid (*e.g.,* it does not contain double-spent transactions). Once a full node validates a block, it relays its contents to peers [74] based on specifications of an underlying block relay protocol such as the *normal*, *compact*, or *Graphene* block relay protocols explained in greater detail in **Section 2.2.4**. As such, full nodes and block relay protocols play a crucial role in the performance and security of the blockchain system [75–78].

Churn of full nodes may occur due to a variety of reasons, such as the need to apply software patches or intermittency of power or network connectivity. Indeed, power outages are common in developing countries [79–86] and not unusual in developed countries as well [87–98]. Churn in different peer-to-peer networks has been widely studied, characterized and modeled [99–105]. However, before the work presented in this thesis, churn had received little attention in relation to blockchains [49, 106–112]. Works that *do* consider churn in their models have not sought a full characterization and evaluation of its impact.

It becomes important, in this context, to empirically measure the occurrence of natural phenomena, such as churn and orphan transactions, in blockchain systems since they can impact the performance of the system. Moreover, metrics including the effects of the aforementioned phenomena on the propagation delay and the communication cost of relaying blocks and transactions must be quantified. While some works study such metrics on *end-to-end* blockchain networks, [40, 45, 110, 113–118], *i.e.,* from the vantage point of the entire network as whole, to the best of our knowledge, very little work has been done to study performance metrics *in-situ* within nodes participating in the blockchain ecosystem [119, 120]. Though the former provide interesting insights into the *overall* system, it is necessary to understand the effects of natural phenomena – such as churn and orphan transactions – that may occur in blockchain systems on full nodes from *their* vantage point since they are crucial for the security of the system. Full nodes exchange a large number of messages amongst one another to relay information such as announcements of new blocks, relay of transactions, and so on. It should thus be useful to extract meaningful data out of these exchanges to help quantify the metrics mentioned above. However, implementations of blockchain protocols are often complex, and significant effort may be required to devise a scheme that enables extraction of useful data.

In the rest of this section, we first identify the main research questions which this thesis seeks to answer. Next, we explain, in detail, the most significant contributions of the thesis. Finally, we summarize key takeaways from this work.

Research questions

In this thesis, we identify the following research questions based on the discussion above.

- *How can one precisely quantify performance metrics such as block propagation delays and the impact of orphan transactions from the vantage point of full nodes in the blockchain system?*
- *Does there exist churn in blockchain networks? If yes, to what degree? How does it affect the performance of the corresponding blockchain system?*
- *Can we mitigate the effect of churn on the performance of a blockchain system?*
- *How do different block relay protocols compare to one another? How does their performance match in realistic network conditions?*
- *What are the circumstances under which transactions may become orphan in a blockchain system? What are the characteristics of these orphan transactions?*
- *How do orphan transactions impact the performance of the underlying blockchain system? Can we mitigate this impact on performance of the system?*
- *Does there exist a relationship between churn of full nodes in a blockchain system and transactions received by the nodes becoming orphan?*

We seek the answers to these research questions through empirical methods. Therefore, we conduct measurement campaigns on popular blockchain systems namely

Bitcoin and an implementation of one of its variant – *Bitcoin Unlimited*. Our contributions and main findings in this thesis are as explained next.

Contributions

The contributions and findings in this thesis are explained in detail as follows.

Churn in the Bitcoin network. Our goal in **Chapter 3** is to systematically characterize churn in the Bitcoin network and answer the following research questions: does there exist churn in the Bitcoin system? If yes, to what extent? What is its impact on the system? How can this impact be evaluated? We describe the methodology we employed to answer these questions. We believe the same methodology can be used in *any* blockchain system to study the effects of churn on the system.

Our characterization of churn in the Bitcoin network is based on measurements of the duration of time that nodes in the network are continuously reachable (*i.e., up session lengths*) and continuously unreachable (*i.e., down session lengths*). For this purpose, we collect data on behavior of nodes in the network. Our data shows that out of more than 40,000 unique nodes on the network, over 97% leave and rejoin the network multiple times over a time span of about two months. In fact, the average churn rate in the Bitcoin network exceeds 4 churns per node per day. Our statistical analysis in the chapter identifies the *log-logistic distribution* and the *Weibull distribution* as the best fits for up session lengths and down session lengths, respectively, among several popular distributions. Next, we analyze churn at the level of IPv4 subnetworks (subnets). Over the measurement period, we find that IP addresses associated with full nodes in the Bitcoin network belong to about 29,000 unique IPv4 /24 subnets. Our analysis further shows that for over 99% of these subnets, fewer than 10 unique IP addresses from each subnet appear on the Bitcoin network over the span of two months. Lastly, we analyze churning behavior of the 10 subnets with the largest numbers of nodes. While churning behavior of nodes within subnets may be correlated, churning behavior of nodes belonging to different subnets appears largely uncorrelated.

The *compact block* relay protocol [54] (described in greater detail in **Section 2.2.4**) is implemented in Bitcoin to compress the sizes of blocks and speed up their propagation. In this context, our next contribution is to empirically evaluate the performance of the protocol under realistic node churning behavior, leveraging our statistical characterization of churn to generate up and down session lengths from the distributions mentioned above. But how can we evaluate this performance?

To this end, we have created a novel *log-to-file* system, which works as follows: we analyze the Bitcoin software and identify events of interest such as arrival of a block, the transactions it contains, and so on. Next, data relevant to each of these events is dumped to files along with nanosecond-precision timestamps that allow us to precisely figure out *when* these events take place. We have made the log-to-file system and all logs generated from experiments public for use by the wider research community [121, 122].

We use the aforementioned generated session lengths to emulate churn on nodes under our control in the *live Bitcoin network* (*i.e.,* on the Bitcoin `mainnet`), taking these nodes off the network and bringing them back on according to the sampled session lengths over a two-week period. Our analysis, compared against a control group of nodes that are continuously connected to the network, shows that the performance of the compact block protocol significantly degrades in the presence of churn. Specifically, the churning nodes see a significantly larger fraction of incomplete blocks as the control nodes (an average of 33.12% vs. 7.15% unsuccessful compact blocks). This is due to an absence of about 78 transactions on average for the churning nodes, versus less than 1 transaction for the control nodes. The end result is that, on average, churning nodes require over five times as much time to propagate a block than their continuously connected counterparts (*i.e.,* 566.89 ms vs. 109.31 ms). The largest propagation delay experienced by blocks received by churning nodes is more than twice the largest propagation delay experienced by any block received by the control nodes (*i.e.,* 105.54 s vs. 46.14 s). These results confirm that churn can have a significant impact on block propagation in Bitcoin. Note that throughout this document, we refer to the single-hop block propagation delay, *i.e.,* the time it takes to completely recover and reconstruct a block once a node receives an announcement of the block from a peer, as *block propagation delay* or *propagation delay.*

To alleviate the aforementioned issues with churning nodes, we propose and implement into the Bitcoin Core a synchronization protocol, dubbed `MempoolSync`, in which a node periodically transmits top-ranked transactions of its mempool to its peers. The goal of the protocol is to provide churning nodes with those transactions that they may have missed during their down times and which are likely to be included in future blocks. Our experimental results indicate that churning nodes that accept `MempoolSync` messages are able to successfully reconstruct, on average, a larger fraction of compact blocks than churning nodes that do not accept such messages (*i.e.,* 83.19% vs. 66.88%). As a result, churning nodes that accept `MempoolSync` messages have significantly smaller block propagation delays on average (*i.e.,* 249.06 ms vs. 566.89 ms). These results show that a scheme that synchronizes mempools of churning nodes with mempools of other highly connected nodes in the Bitcoin network may be able to overcome performance degradation issues.

Comparison of block relay protocols. As noted earlier, there exist multiple block relay protocols in different blockchain systems. Intuitively, one may ask: how do these protocols compare one to another in real-world network conditions?

We answer this question in **Chapter 4**. We analyze and quantify real-world effects on three popular block propagation protocols used in the Bitcoin ecosystem (described in greater detail in **Section 2.2.4**): (*i*) the legacy (*default*) block propagation protocol [1], (*ii*) the *compact block* relay protocol [54], and (*iii*) the more recent *Graphene* block relay protocol [55, 123]. Our comparisons are carried out through the popular Bitcoin Unlimited (BU) client, which is a concrete implementation for Bitcoin Cash (a fork of the Bitcoin blockchain - see **Section 2.2.2**) that can support all three protocols, after some code changes. Unlike existing simulation-based evaluations, our experiments include important real-world artifacts such as the fluctuations in node connectivity that are ubiquitous for these networks.

Our experimental testbed, which consists of 12 full nodes, operates in three network regimes. The first regime represents an ideal situation where the full relay nodes in our testbed are `always on`, *i.e.,* continuously connected to the BU network. In the second regime, nodes in the testbed exhibit `statistical churn`, mimicking the statistical behavior of real-life churning nodes on the Bitcoin network [68–70]. In the third regime, nodes experience `periodic churn`, wherein nodes cycle through "on" and "off" periods at a fixed frequency. This regime allows us to compare and contrast the impact of different churn parameters on the performance of block relay protocols, and emulate the aforementioned electricity outages. We stress that all our experiments consider full nodes that relay blocks, but do not mine them.

We extend our log-to-file system to provide fine-grained measurement capabilities to assess the performance of the three aforementioned block relay protocols. We are now able to not only record events, such as arrival of blocks, but also very precise information that enables us to provide in-depth analyses of the block relay protocols. We have made this version of our log-to-file system as well as relevant experimental logs publicly available for use by the wider research community [124, 125].

Our experiments show that for nodes following the `always on` and `statistical churn` regimes, the Graphene block relay protocol performs by far the best. Indeed, the mean propagation delays for the compact and the legacy blocks are 1.4 times and 5 times larger, respectively. Similarly, the mean communication sizes are respectively $1.8 - 1.9$ times and $15 - 25$ times larger. In nodes following the `periodic churn` regime, the Graphene block relay protocol still generally performs best, but there are cases where the compact block relay protocol is superior. In particular, this happens when nodes churn frequently and miss receiving transactions that are included in the blocks received by the nodes soon after they rejoin the network. We found there to be a moderate to high correlation between the amount of communication needed to successfully reconstruct blocks and the propagation delay of blocks, regardless of the location of the peers of our testbed nodes. This indicates that as the amount of communicated required to successfully reconstruct blocks increases, so does the delay in propagation of the respective block.

Next, we perform an in-depth analysis of the Graphene block relay protocol. In particular, we examine the cases in the `periodic churn` regime where the performance of the protocol degrades. The analysis shows that when nodes churn frequently, they more often require as many as two additional round-trip communications to recover information needed to successfully reconstruct blocks which also adds to the delay in the propagation of the block. A temporal analysis of the protocol in both the `statistical churn` and `periodic churn` regimes shows that a higher fraction of blocks received immediately after nodes rejoin the network contain transactions that are missing from their mempools. However, this fraction quickly falls as the nodes stay connected to the network for long.

Finally, we further compare the efficiency of the three protocols by examining the sizes of the first messages received by the nodes from their peers. Our analysis shows that the sizes of the first messages received in the legacy block relay protocol are almost always larger than first messages received in the remaining two protocols. Indeed, first messages received in the Graphene and compact block relay protocols are respectively, on average, roughly 50 times and 10 times smaller. We find that the sizes of the first messages received in the compact block relay protocol correlated with the number of full (additional) transactions included in the message. Further, excluding the coinbase transaction, full transactions contained in the first message sent by the compact block protocol are often insufficient or redundant for block decoding. There is not a single case where the first message contains at least two full transactions and all such transactions are helpful.

Orphan transactions in the Bitcoin network. In **Chapter 5**, we extend our measurement methodology to better understand the properties and behavior of orphan transactions in the Bitcoin system which remains a largely unexplored field to date. Hence, 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? Does node churn affect orphan transactions? To the best of our knowledge, there exists no work that comprehensively answers these questions.

Our first goal in this context 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.

Next, we study the impact of network and performance overhead caused by orphan transactions. We 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 1,000 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.

Finally, we analyze 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.

Takeaways

Full nodes play a crucial role in guaranteeing consensus in blockchains (*i.e.,* validating and relaying transactions and blocks). This thesis fills a gap in terms of measuring the performance of blockchains – in particular Bitcoin and its variant Bitcoin Cash – from the vantage point of full nodes. We achieve this by designing an experimental infrastructure consisting of a measurement software and a testbed. Using this experimental infrastructure, the thesis unveils and quantifies significant inefficiencies in relay mechanisms currently in use in Bitcoin and Bitcoin Cash. The thesis proposes, implements, and evaluates several methods to address these inefficiencies. In summary, the main contributions in the thesis are as below.

- We develop fine-grained measurement capabilities to log and assess the performance of Bitcoin and Bitcoin Cash blockchains *in-situ*.

- We confirm that there exists churn in the Bitcoin network. We show that as many as 97% of more than 40,000 nodes in the Bitcoin network churn *at least* once in a 60-day time-frame. We next systematically characterize this churn.

- We experimentally evaluate the compact block relay protocol under realistic churn. We show empirically that the performance of the protocol degrades whereby blocks miss more transactions and incur larger propagation delays.

- We propose, implement, and evaluate a synchronization protocol as a proof-of-concept to alleviate observed issues and to highlight the benefits of synchronizing mempools of churning nodes with highly-connected nodes. We find that doing so reduces the number of transactions missing in blocks consequently decreasing the propagation delay of blocks.

- We set-up a testbed and empirically compare the performance of three protocols – namely the *default*, *compact*, and *Graphene* block relay protocols – implemented in Bitcoin Unlimited under three different network regimes through extensive experiments. We find that the Graphene block relay protocol generally performs best in the `always on` and `statistical churn` regimes whereas the compact block relay protocol may perform better in some cases in the `periodic churn` regime.

- We perform an in-depth analysis of the dynamic and temporal behavior of the protocols, through which we identify inefficiencies. In particular, we find that nodes may need to perform multiple round-trips of additional communication to successfully reconstruct Graphene blocks. In the case of the compact block relay protocol, we discover that the full transactions included in the first block messages are either not useful at all or not enough to be completely useful in reconstructing blocks.

- We study the circumstances that may make Bitcoin transactions orphan. We observe that counter-intuitively orphan transactions tend to have *fewer* parents when compared to non-orphan transactions. The parents of orphan transactions usually have smaller fees, larger sizes, and smaller fee per byte when compared to parents of non-orphan transactions.

- We show that the same transaction may be added to the orphan pool several times due to getting evicted when the pool becomes full. We conduct experiments to show that this behavior causes unnecessary network overhead since the same transaction may be received multiple times from peers. Yet, we show that by slightly increasing the size of the orphan pool from the default capacity of 100 transactions to 1,000 transactions, the network overhead can be significantly reduced with negligible performance overhead. Increasing the timeout of orphan pool eviction, however, does not have any positive effects.

- We examine the effect of churn on transactions becoming orphan. We find that roughly 25% of transactions received by the node immediately after it rejoins the network become orphan. However, as the node stays connected to the network for long, this fraction reduces significantly. In addition, roughly 50% of orphan transactions arrive within the first two hours after a churning node rejoins the network. We find that orphan transactions repeatedly get evicted from the orphan pool in the first two hours. Therefore, it may be beneficial for churning nodes to be configured with a larger orphan pool size to minimize unnecessary network overhead.

Road map

The rest of this thesis is organized as follows. In **Chapter 2**, we provide background information on concepts required to understand the content of the following chapters. We also survey literature relevant to our work. We introduce our novel log-to-file system, characterize churn in the Bitcoin network, study its impact on the performance of the protocol, and present and evaluate a novel synchronization scheme to mitigate the harmful effects of churn on the Bitcoin protocol in **Chapter 3**. We present a comparison of performance of three different block relay protocols in realistic network conditions in **Chapter 4** as well as in-depth analyses of the compact and Graphene block relay protocols. In **Chapter 5**, we study circumstances that make transactions orphan and properties of their parents, the impact of varying default parameters related to the orphan pool, and the relation between churn in the Bitcoin network and transactions received by a churning node that become orphan. **Chapter 6** discusses possible future works and concludes the thesis.

Chapter 2

Background and related work

In this chapter, we first take a look at the blockchain technology. We explain in detail the underlying components of a blockchain in **Section 2.1**. Next, in **Section 2.2**, we describe in particular the Bitcoin blockchain and its variant, Bitcoin Unlimited. We also outline the different block relay protocols employed by these blockchain systems. In **Section 2.3**, we give an overview of orphan transactions. This background information forms the basis for understanding the work and results presented in **Chapters 3** to **5**. Finally, we survey the literature relevant to the work presented in this thesis in **Section 2.4**.

2.1 Preliminaries

We give background information on the basic underlying components of a general blockchain in this section.

2.1.1 Transaction

A *transaction* is simply a transfer of token such as cryptocurrency, digital assets, etc., from one or more source accounts to one or more destination accounts. An *account* in a blockchain system is usually a public-private key pair. The public key identifies the entity to whom tokens are transferred whereas the holder of the account signs the transaction transferring the aforementioned token with their private key.

A transaction may have multiple inputs and multiple outputs. The inputs of a

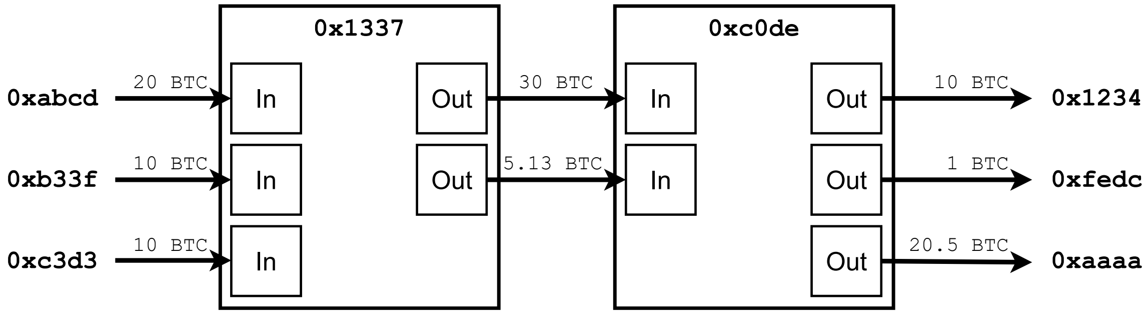


Figure 2·1: A Bitcoin transaction with multiple input accounts and multiple output accounts. The difference between the total input value and the total output value is the transaction fee.

transaction are outputs of a *parent* transactions each of which can only be claimed once so as not to spend the same token several times. The sum of the values of the inputs must be greater than or equal to the sum of the values of the outputs. The difference between the sum of the values of the inputs and the sum of the values of the outputs is called the *transaction fee*. Each transaction has a unique identifier.

Figure **2·1** shows an illustration of a couple of Bitcoin transaction with unique identifiers (*i.e.,* `0x1337` and `0xc0de`). The transaction on the right (*i.e.,* with identifier `0xc0de`) spends from the outputs of the transaction on the left (*i.e.,* with identifier `0x1337`). The latter has input values that sum to 40 `BTC` and output values that sum to 35.13 `BTC` where the difference between the two sums, *i.e.,* 4.87 `BTC`, is the transaction fee. Similarly, the former has input values that sum to 35.13 `BTC` from which it spends 31.5 `BTC` with the remaining amount, *i.e.,* 3.63 `BTC`, paid as the transaction fee.

2.1.2 Block

A *block* is a collection of transactions that have not yet appeared in any of the previous blocks. It is the record keeping mechanism of a blockchain system. Each block usually contains (*i*) a header that contains metadata about the block, and (*ii*) the actual transactions [126–128]. The block identifier is simply the hash of the contents of the

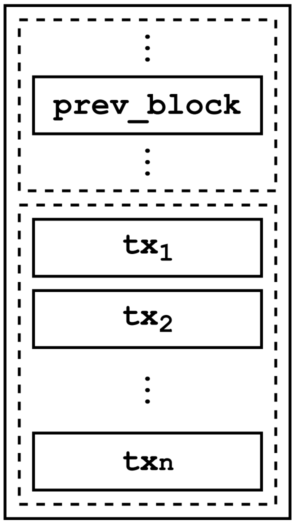


Figure 2·2: An illustration of a block.

block. In Bitcoin, a new block is created, on average, every 10 minutes. **Figure** 2·2 shows an illustration of a block.

Transactions are included in topological order in blocks [129]. Suppose that there are two transactions, $\mathtt{tx}_\mathtt{A}$ and $\mathtt{tx}_\mathtt{B}$, and that the latter spends from the former. Then, $\mathtt{tx}_\mathtt{A}$ must either appear in a previously created block, or in an order such that $\mathtt{tx}_\mathtt{A}$ appears before $\mathtt{tx}_\mathtt{B}$ in the same block.

In Bitcoin, a *merkle tree* [130] is created by placing transactions as leaves of a binary tree and then repeatedly concatenating and hashing their identifiers together. The root of the binary tree is called the *merkle root*. It is placed inside the header of the block so that any recipient of the block can verify the validity of the transactions in the block. **Figure** 2·3 shows an illustration of a merkle tree. Different blockchains may use variants of the merkle tree [131] or another scheme altogether [132].

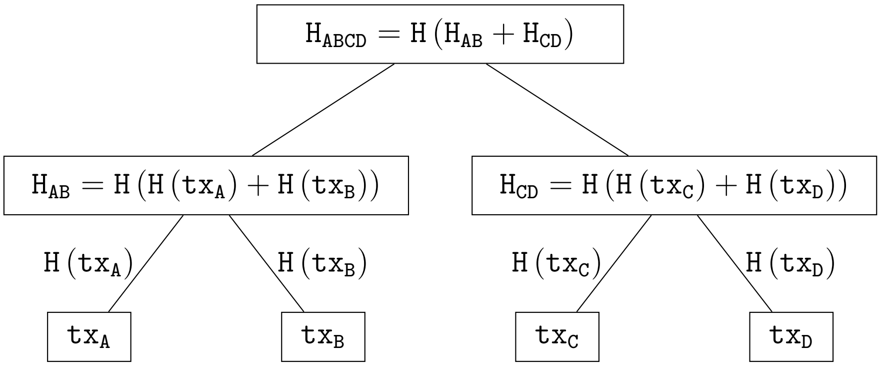

$\mathtt{H}_\mathtt{ABCD} = \mathtt{H}(\mathtt{H}_\mathtt{AB} + \mathtt{H}_\mathtt{CD})$
$\mathtt{H}_\mathtt{AB} = \mathtt{H}(\mathtt{H}(\mathtt{tx}_\mathtt{A}) + \mathtt{H}(\mathtt{tx}_\mathtt{B}))$
$\mathtt{H}_\mathtt{CD} = \mathtt{H}(\mathtt{H}(\mathtt{tx}_\mathtt{C}) + \mathtt{H}(\mathtt{tx}_\mathtt{D}))$
$\mathtt{H}(\mathtt{tx}_\mathtt{A})$
$\mathtt{H}(\mathtt{tx}_\mathtt{B})$
$\mathtt{H}(\mathtt{tx}_\mathtt{C})$
$\mathtt{H}(\mathtt{tx}_\mathtt{D})$
$\mathtt{tx}_\mathtt{A}$
$\mathtt{tx}_\mathtt{B}$
$\mathtt{tx}_\mathtt{C}$
$\mathtt{tx}_\mathtt{D}$

Figure 2·3: An illustration of a merkle tree created by transactions $\mathtt{tx}_\mathtt{A}$, $\mathtt{tx}_\mathtt{B}$, $\mathtt{tx}_\mathtt{C}$, and $\mathtt{tx}_\mathtt{D}$. The node with the value $\mathtt{H}_\mathtt{ABCD}$ is called the *merkle root* of the tree.

2.1.3 Miner

Mining is the process of packing transactions into blocks. A *miner* is a participant of the blockchain network that gathers transactions to put inside a block. Once included in a block, a transaction becomes finalized and irreversible as more blocks are *mined* on top of the block containing it.

Miners are usually incentivized to stay honest by providing to them monetary benefits: every time a new, valid block is created, the miner who created the block is rewarded with new minted cryptocurrency. For example, at the time of writing, in Bitcoin, miners are rewarded 6.25 BTC for mining a new block. In addition, miners can keep fees for all transactions included in the block that they mine.

To make sure other participants of the network trust and accept the blocks created by miners, the latter must follow a *consensus protocol* to mine a new block. At present, there are several consensus protocols that exist in the blockchain space [133–135], *e.g., proof of work* [1], *proof of stake* [136, 137], *proof of elapsed time* [138], etc. Bitcoin follows the proof of work consensus protocol which requires miners to solve a computationally expensive mathematical problem to prove their honesty.

2.1.4 Blockchain

A *blockchain* is a distributed, decentralized ledger that records a history of transactions that take place in an associated network. It is a chronologically ordered collection of blocks that are chained together with cryptographic hashes of previous blocks. This makes the blockchain *immutable*, *i.e.,* it is practically impossible (as long as the underlying cryptography is secure) for an adversary to tamper with the data recorded in the blockchain without controlling a significant amount of resources.

Conventional scheme such as transferring money between two parties usually involves a centralized entity such as a bank. Blockchains, on the other hand, are

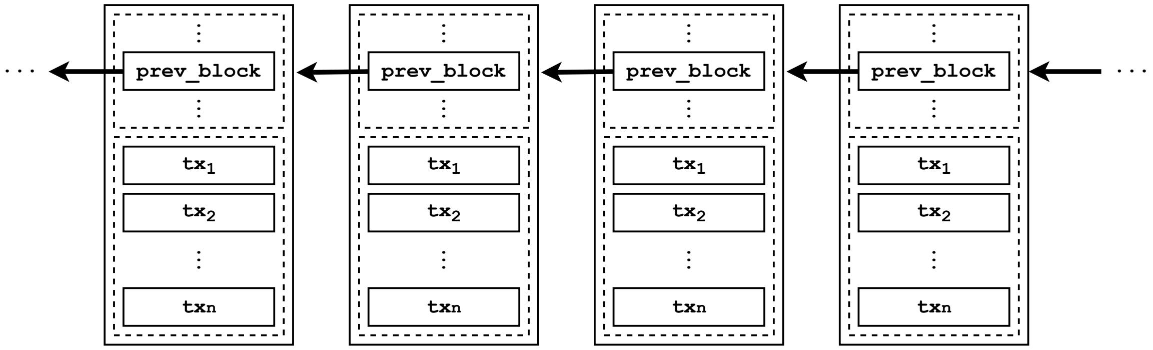


Figure 2·4: An illustration of a blockchain. Each block is cryptographically linked to the previous creating an unalterable chain of blocks.

decentralized and reduce the amount of trust placed in a single actor. Instead they require an entire network of users to establish and maintain consensus who are often provided economic incentives to remain honest.

A blockchain may be *permissionless* where anybody can view the records embedded in the blockchain and any entity with enough resources can mine a new block. On the other hand, a blockchain may be *permissioned*, incorporating an additional access control layer and, thereby, limiting who can read from or append to the blockchain. Bitcoin's blockchain is permissionless, *i.e.,* anyone can read from it and write to it if they have enough resources to solve a complex mathematical problem (see **Section 2.1.3**). ConsenSys' Quorum blockchain [139], on the other hand, is permissioned allowing only certain authorized entities to access and/or modify it. **Figure 2·4** illustrates a blockchain.

2.2 Bitcoin and Bitcoin Unlimited

In this section, we first comprehensively introduce the Bitcoin blockchain and then give an overview of an implementation – called Bitcoin Unlimited – of its variant Bitcoin Cash. Next, we explain the data structures employed in block relay protocols which we also describe in greater detail.

2.2.1 Bitcoin

The Bitcoin cryptocurrency, originally introduced by Satoshi Nakamoto in 2008 [1] as a peer-to-peer electronic payment system, is currently used for buying and selling a wide variety of goods in different markets across the globe including Virgin Galactic [33], AT&T [34], Newegg [35], and more. At the time of writing, the cryptocurrency has a total market capitalization of well over *one trillion* USD [36]. Bitcoin uses a blockchain to record all transactions that take place in the Bitcoin network [45]. Each new transaction is broadcast over the network, and thereafter recorded by every node in its local memory pool (known as a *mempool*) for subsequent consensus-based validation. Transactions stay in the mempool until confirmed by a Bitcoin miner [140] and added to a block in Bitcoin's blockchain. Every day, hundreds of thousands of transactions are created and confirmed in the Bitcoin network [141], resulting in close to 690 million transactions (at the time of writing) since its inception [142]. By design, a new block containing transactions is created by a miner and propagated over the network's nodes on average once every ten minutes [126].

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* (see **Section 2.3**), 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 in **Chapter 5**.

Bitcoin employs the *proof-of-work* [1] algorithm to maintain consensus in the system whereby a miner must do some work to solve a complex mathematical problem in order to create a new block which can be added to the blockchain. The amount of work that a miner needs to do is determined by the current network *difficulty* [143] which determines how difficult it is to create a new block. It depends on the total mining power of the network which is close to 160×10^{18} hashes per second at the time of writing [144].

Once a miner creates a new block, it is propagated to other nodes in the Bitcoin network via the *default* and *compact* block relay protocols which we explain in greater detail in **Section 2.2.4**. We study the performance of the compact block relay protocol extensively in **Chapter 3**.

2.2.2 Bitcoin Unlimited

Bitcoin Unlimited is an implementation of the Bitcoin Cash (BCH) protocol which was forked from the reference implementation of Bitcoin, also known as Bitcoin Core and first released in 2016 [145]. A natural question to ask is: why did there arise a need to fork Bitcoin Core?

The problem of scalability in Bitcoin is quite well known [146]. The original Bitcoin protocol placed a 1 MB size limit on a block. While the exact reason remains unknown, it is speculated that Satoshi Nakamoto placed this limit to prevent adversaries from creating very large blocks filled with invalid transactions and spamming the network [147]. This resulted in the maximum rate at which transactions could be committed to the blockchain at roughly 7 transactions per second [47] which was sufficient at the time. However, over the years, as the popularity of Bitcoin grew, so did the volume of transactions. In 2010, for example, the number of transactions confirmed, *i.e.,* added to a block, per day was recorded at less than 200. In comparison, by 2016, this number grew to around 250,000 confirmed transactions per day - a roughly 3 orders of magnitude increase [148]. However, roughly 25,000 transactions still remained unconfirmed, *i.e.,* waiting to be added to a block, per day. Critics of Bitcoin Core believe that the block size limit is one of the reasons to blame for the low throughput of transaction confirmation in Bitcoin.

Enters BCH-compatible Bitcoin Unlimited. It differs from Bitcoin Core mainly in the following ways: a) the block size limit is removed, and b) miners can freely adjust the block size [149]. Bitcoin Unlimited promises a higher transaction throughput than Bitcoin Core. While the number of transactions in a Bitcoin Core block remains strictly below 2,600 [150], Bitcoin Unlimited blocks have been known to contain as many as a little over 24,000 transactions[1] [152]. However, with a much smaller transaction volume than Bitcoin Core in the present day, the full potential of Bitcoin Unlimited still remains to be seen.

Similar to Bitcoin Core, a new block is generated, or *mined*, roughly every 10 minutes on average in Bitcoin Unlimited. Likewise, a new transaction when announced is stored in the local memory, dubbed *mempool*, of every node participating in the network where it remains until added to a future block. In addition to the default and compact block relay protocols, Bitcoin Unlimited also implements the *Graphene* block relay protocol which explain in greater detail in **Section 2.2.4**. We empirically evaluate the performance of the three block relay protocols in **Chapter 4**.

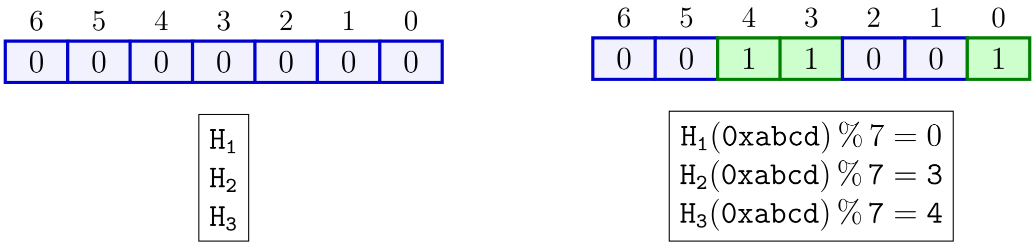
2.2.3 Data structures

We now introduce and explain the underlying data structures employed in the Graphene block relay protocol which is implemented in Bitcoin Unlimited. We describe the aforementioned protocol in greater detail in **Section 2.2.4**.

Bloom filter. A *Bloom filter* [153] is essentially a bit-vector, *i.e.,* an array A of size m where each element in the array can only be either 0 or 1. Given a set S of n objects, the Bloom filter allows testing an object s for membership in S. The bit-vector A representing a Bloom filter is first initialized such that all bits are set to 0, *i.e.,* $\forall a \in [1, m], A[a] = 0$. To represent S as a Bloom filter, each object $s_i \in S$, where $i \in [1, n]$, is hashed with predefined hash functions $h_1, h_2, \ldots, h_K$. The output

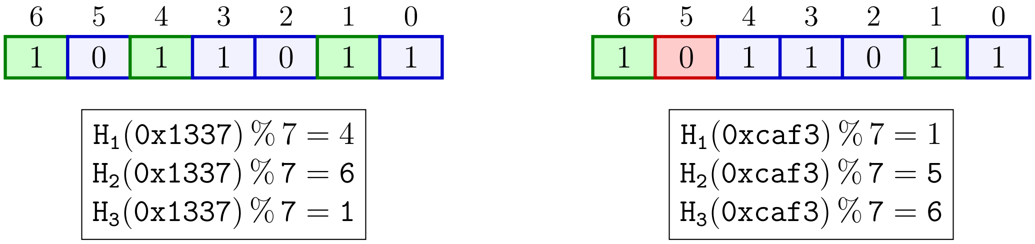
---

[1] For example, block height: 681765 [151].



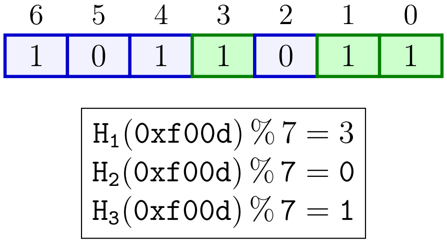
(a) Empty Bloom filter. All elements are initialized to 0. Hash functions `H`$_1$, `H`$_2$, and `H`$_3$ map an item to locations in the Bloom filter.

(b) Bloom filter after inserting `0xabcd` into the filter. Elements at locations obtained as a result of the hash functions `H`$_1$, `H`$_2$, and `H`$_3$ (shown in green background) are set to 1.



(c) Bloom filter after inserting `0x1337` into the filter. Elements at locations obtained as a result of the hash functions `H`$_1$, `H`$_2$, and `H`$_3$ (shown in green background) are set to 1.

(d) Lookup for `0xcaf3` which is *definitely* not a member of the Bloom filter since not all elements at locations obtained as a result of the hash functions `H`$_1$, `H`$_2$, and `H`$_3$ (shown in red and green backgrounds) are set to 1.



(e) Lookup for `0xf00d` which is *probably* a member of the Bloom filter since all elements at locations obtained as a result of the hash functions `H`$_1$, `H`$_2$, and `H`$_3$ (shown in green background) are set to 1. Note that this is a false positive since `0xf00d` was never inserted in the Bloom filter.

Figure 2·5: Illustration of Bloom filter operations.

of each hash function determines a location in the bit-vector, the bit corresponding to which is set to 1, *i.e.*, $\forall i \in [1, n], \forall k \in [1, K], A\left[h_k(s_i)\right] = 1$. The object r that is to be tested for membership in S is hashed against the same hash functions. r is *most likely* a member of S if *all* bits corresponding to the locations in A obtained by hashing it are set to 1, *i.e.,* when the result of

$$\bigcap_{k \in [1,K]} A\left[h_k(r)\right]$$

is 1. r is *definitely* not a member of S even if a single bit corresponding to the locations in A obtained by hashing it is set to 0, *i.e.,* when the result of

$$\bigcap_{k \in [1,K]} A\left[h_k(r)\right]$$

is 0.

The Bloom filter is a *probabilistic* data structure. Given a set of objects S and a Bloom filter A of size m, denote with $\mathbb{P}_{fp}$ the probability that a false positive and with $\mathbb{P}_{fn}$ the probability that a false negative occurs as a result of the test of membership of an object r in S. While a Bloom filter *does* allow false positives, *i.e.,* $\mathbb{P}_{fp} \geq 0$, false negatives can *never* occur, *i.e.,* $\mathbb{P}_{fn} = 0$. Further, $\mathbb{P}_{fp}$ is configurable and depends on the number of objects in the set, *i.e.,* $|S|$, the size of the Bloom filter, *i.e.,* m, and the number of predefined hash functions, *i.e.,* K [154, 155]. **Figure 2·5** shows an illustration of Bloom filter operations.

Invertible Bloom lookup table. A Bloom filter does not allow listing of all members of the set that it represents. This operation can be performed with an *invertible Bloom lookup table (IBLT)* [156, 157] which, in addition to the insertion and lookup operations, also supports listing members of a set.

Given two sets of objects S_1 and S_2 where the contents of the sets are not identical, IBLTs can be used to efficiently find the symmetric differences between the two sets. We represent the two sets S_1 and S_2 each with an IBLT I_{S_1} and I_{S_2}, respectively. The symmetric difference between the two sets, *i.e.,* $D_{S_1-S_2}$ or, conversely, $D_{S_2-S_1}$, can

Listing 1: Workflow of the default block relay protocol. `SRC` and `DST` are peers where the former sends a default block which the latter receives.

```
SRC: send headers to DST for block $G$
DST: request block $G$ from SRC via getdata
SRC: send block $G$ to DST via block
DST: process block $G$ and relay to peers
```

then be found by first subtracting the IBLTs from one another using a process dubbed as the *peeling* process, *i.e.*, $I' = I_{S_1} - I_{S_2}$, and then finally decoding the difference I' [158]. The decoding process is *successful* if all symmetric differences between S_1 and S_2 are found. Otherwise, the decode process has *failed.* Similar to a Bloom filter, an IBLT is a probabilistic data structure, and there is a non-zero probability that the decode process will fail. When this happens, the process can be repeated by modifying the parameters that govern the size of the IBLTs.

2.2.4 Block relay protocols

When a node in the Bitcoin network receives a new block from one of its peers, it processes the block and relays it forward to its remaining peers. This relay of blocks allows the trust-less Bitcoin network to maintain consensus on valid transactions and balances available in wallets (or user accounts). We describe next the three block relay protocols of interest that are implemented in Bitcoin and Bitcoin Unlimited which we evaluate empirically in **Chapters 3** and **4**.

For the purpose of explaining the protocols, we consider two nodes `SRC` and `DST` that are connected to one another. `SRC` is a source node that relays information to `DST` which is the destination node.

Default (normal) block relay. The original block relay protocol implemented by Satoshi Nakamoto in Bitcoin is the *normal block relay* protocol [1] or the *default block relay* protocol. In this protocol, blocks are relayed with full transactions included

Listing 2: Workflow of the compact block relay protocol. `SRC` and `DST` are peers where the former sends a compact block which the latter receives.

```
SRC: send headers to DST for block $G$
DST: request block $G$ from SRC via getdata
SRC: send block $G$ to DST via cmpctblock
DST: attempt to reconstruct block $G$
if reconstruct successful, i.e., no missing transactions in block $G$ then
    DST: process block $G$ and relay to peers
else
    DST: request missing transactions from SRC via getblocktxn
    SRC: send requested transactions to DST via blocktxn
    DST: reconstruct block $G$
    DST: process block $G$ and relay to peers
```

which often results in a waste of bandwidth since the receiving node most likely already received these transactions from its peers earlier. **Listing 1** shows the process of the default block relay. `SRC` announces the availability of a new block by sending the block's header to `DST` via a `headers`[2] message. `DST` requests the block from `SRC` if it is not already present in its inventory by sending a `getdata` message to the latter. `SRC` then responds with the full block via the `block` message. `DST` processes the block which includes, among other steps, validating the block header, validating the transactions within the block, removing transactions included in the block from its mempool, and so on. `DST` then propagates the block to its other peers.

Compact block relay. The *compact block relay* protocol was introduced to Bitcoin Core in 2016 [54] and implemented in Bitcoin Unlimited in 2019 [159]. Unlike a default block which contains full transactions, the compact block only contains a 6-byte hash of each transaction with only a few full transactions (including the coinbase transaction). We evaluate the usefulness of these extra full transactions in **Section 4.2.4**. This process reduces the bandwidth required to propagate a block

---

[2] Note that newer versions of Bitcoin Unlimited replace the legacy `inv` message with the `headers` message for the purpose of block relay.

by several orders of magnitude. The protocol works under the assumption that the receiver of the block already has in its mempool all transactions that are representative of the 6-byte hashes contained in the block.

Listing 2 shows the relay of a compact block. `SRC` announces the availability of a new block by sending a `headers` message to `DST` who requests the block using the `getdata` message if the block is not already present in its inventory. `SRC` responds with the `cmpctblock` message which contains the block header, some full transactions, and 6-byte hashes of the remaining transactions. If no transactions that correspond to the 6-byte hashes in the block are *missing* from its mempool, `DST` is able to successfully *reconstruct* the block, and propagate it to peers. This scenario is represented by lines 5-6 in the listing.

If `DST` cannot find all transactions in its mempool that correspond to the 6-byte hashes that are included in the block, the compact block reconstruction *fails*. This is represented by lines 7-11 in **Listing 2**. `DST` requests these missing transactions by sending a `getblocktxn` message to `SRC` who responds with the requested transactions in a `blocktxn` message. `DST` is now able to successfully reconstruct the block and propagate it forward to peers. It is evident that recovering missing transactions requires extra round-trip communication which incurs delay in propagation of the block and consumes additional bandwidth.

Note that, to the best of our knowledge, unlike Bitcoin Core that supports both *low bandwidth* and *high bandwidth* modes in the compact block protocol, Bitcoin Unlimited only supports the low bandwidth mode [160] whereby a node only propagates a block forward after fully validating it locally. We refer the reader to the documentation of the compact block protocol for further details on the two modes [54]. **Listing 2** depicts the low bandwidth mode of the protocol. We emphasize, however, that regardless of the mode of operation chosen by two peers to exchange information in, if a receiver's mempool contains all the transactions whose hashes are contained in a compact block that it received, only then will it be able to successfully reconstruct the original block. However, if not all transactions are already in the node's mempool then it will fail to reconstruct the block resulting in additional consumption of network resources.

Graphene block relay. The *Graphene block relay* [55, 123] protocol is a more complex protocol that uses probabilistic data structures, namely Bloom filters and invertible Bloom lookup tables (IBLTs) to relay blocks (see **Section 2.2.3** for background on both). The combination of these data structures further reduces the size of the block and, consequently, the bandwidth required to relay blocks.

We next examine the Graphene block relay protocol in detail. **Listing 3** shows the workflow of the Graphene block relay protocol. `SRC` announces the availability of a new block to `DST` via the `headers` message. If `DST` does not already have the block in its inventory, it sends a request to `SRC` with a `get_grblk` message *along with* the size m of its mempool, *i.e.,* the number of transactions in its mempool. `SRC` encodes the hashes of transactions in the block in a Bloom filter B and an IBLT I. It uses m as a parameter to determine the sizes of these data structures which in turn determines the number of symmetric differences that can be recovered from I. `SRC` then sends the Bloom filter B and the IBLT I to `DST` with the `grblk` message. `DST` collects hashes of all transactions currently in its mempool and orphan pool [38, 39] and passes them through the Bloom filter B. It creates a candidate set C of transactions whose hashes successfully pass through the filter. `DST` then uses C to construct a local IBLT I', performs the subtraction operation, *i.e.,* $I - I'$, and attempts to extract the transaction hashes encoded in I by decoding the result obtained from the subtraction operation. If the decoding process is successful *and* `DST` has in its mempool and/or orphan pool all transactions that are included in the

Listing 3: Workflow of the Graphene block relay protocol. `SRC` and `DST` are peers where the former sends a Graphene block which the latter receives.

```
SRC: send headers to DST for block $G$
DST: request block $G$ from SRC via get_grblk
SRC: encode hashes of transactions in block $G$ into Bloom filter $B$ and IBLT $I$
SRC: send $B$ and $I$ to DST in grblk
DST: create transaction candidate set $C$ from transactions in mempool and orphan pool whose hashes pass through $B$
DST: create IBLT $I'$ from $C$
DST: attempt to extract encoded transaction hashes, i.e., find $I - I'$, and decode result
if IBLT subtraction $I - I'$ and decode successful then
    DST: attempt to reconstruct block $G$
    if reconstruct successful, i.e., no missing transactions in block $G$ then    ▷ ①
        DST: process block $G$ and relay to peers
    else    ▷ ②
        DST: request missing transactions from SRC via get_grblktx
        SRC: send requested transactions to DST via grblktx
        DST: reconstruct block $G$
        DST: process block $G$ and relay to peers
else ▷ IBLT subtraction and decode failed; initiate failure recovery
    DST: create Bloom filter $F$ of transaction set $C$
    DST: request failure recovery from SRC by sending $F$ via get_grrec
    SRC: find set $C'$ of transaction that are in block $G$ but not in Bloom filter $F$
    SRC: create IBLT $J$
    SRC: send set $C'$ and IBLT $J$ to DST via grrec
    DST: create IBLT $J$ from candidate set $C \cup C'$
    DST: attempt to extract encoded transaction hashes, i.e., find $J - J'$, and decode result
    if IBLT subtraction $J - J'$ and decode successful then
        DST: attempt to reconstruct block $G$
        if reconstruct successful, i.e., no missing transactions in $G$ then    ▷ ③
            DST: process block $G$ and relay to peers
        else    ▷ ④
            DST: request missing transactions from SRC via get_grblktx
            SRC: send requested transactions to DST via grblktx
            DST: reconstruct block $G$
            DST: process block $G$ and relay to peers
    else    ▷ ⑤
        DST: initiate fail-over mechanism by requesting default block
```

block, it reconstructs and processes the block, and then propagates it to its peers. This scenario is represented by the code branch marked by ① in **Listing 3**.

However, even if the subtraction operation $I - I'$ and the successive decoding operation are successful, there may be some transactions in the block that are missing from the mempool and orphan pool of `DST`. In this case, `DST` needs to recover the missing transactions before it can process the block. `DST` does this by sending a `get_grblktx` message to `SRC` who responds with the requested transactions in a `grblktx` message. `DST` is now able to successfully reconstruct the block, process it, and propagate it to peers. Note that similar to compact block relay, recovering missing transactions requires an extra round-trip communication and delays the propagation of the block in addition to consuming additional bandwidth. This scenario is represented by the code branch marked by ② in **Listing 3**.

Next, we look at the case when the subtraction operation $I - I'$ and the successive decode operation are not successful. This may happen when `DST` is missing too many transactions from its mempool. It, thus, cannot create a transaction candidate set C and, consequently, an IBLT I' that is sufficient to perform the subtraction and decode operations successfully. When this happens, `DST` enters *failure recovery* [161] with `SRC`. This step requires extra round-trip communication to recover from the IBLT decode failure which further delays block propagation and consumes extra bandwidth.

To perform failure recovery, `DST` creates a new Bloom filter F and inserts into it hashes of the transactions from the candidate set C. This enables `SRC` to determine which transactions are present in `DST’s` mempool. `DST` then sends F to `SRC` with a `get_grrec` message. `SRC` creates a new set of transactions C' that is comprised of the transactions that are in the block but whose hashes are not in F. It also creates a revised IBLT J adjusting for the false positives that appear in F, and then sends C' and J to `DST` as a `grrec` message. `DST` creates a new candidate set of transactions

$C \cup C'$. It locally creates an IBLT J' from this candidate set and uses it to extract the transaction hashes encoded in J by first performing the subtraction operation $J - J'$ and then decoding the result of the subtraction operation.

If the subtraction and decode operations are successful, `DST` attempts to reconstruct the block. If at this point, there are no transactions missing from its mempool, `DST` successfully reconstructs the block, processes it, and propagates it forward to peers. This scenario is represented by the code branch marked by ③ in **Listing 3**.

If, however, there are still some transactions missing from its mempool, `DST` requests these transactions from `SRC`. Upon receiving the missing transactions, `DST` successfully reconstructs the block, processes it, and propagates it forward to peers. Note that this scenario requires yet another round-trip communication further delaying block propagation and consuming additional bandwidth. This scenario is represented by the code branch marked by ④ in **Listing 3**.

One may wonder why there may still be missing transactions after failure recovery. This may be because the Bloom filter has a non-zero probability of false positives (see the discussion in **Section 2.2.3**) and `SRC` may falsely conclude that there already exists a transaction in Bloom filter F and not include it in the set of transactions C'. In this case, `DST` may end up in a situation where it needs to perform another round-trip communication to recover transactions that are included in the block but not present in its mempool.

Finally, if the subtraction operation $J - J'$ and the successive decode operation also fail, `DST` must fall back to a fail-over mechanism by requesting a full block (as in the default protocol) instead of a Graphene block from `SRC`. This scenario is represented by the code branch marked by ⑤ in **Listing 3**.

2.3 Orphan transactions

A node in a blockchain network 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 blockchain nor exist in the node's local transaction pool dubbed *mempool*). The node cannot accept the newly received transaction into its mempool until it can verify that the transaction spends valid token, 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*. In Bitcoin, for example, the orphan pool 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.

We characterize orphan transactions in Bitcoin and study their impact on the blockchain system in **Chapter** 5. In this system, 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` [162]):

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 [163].

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* [164] 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 [165], 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.

2.4 Related work

In this section, we survey literature relevant to the work presented in this thesis. We first present commentary on works relevant to existing measurement tools in the blockchain space in **Section 2.4.1**. Next, we review works applicable to block propagation and churn, and orphan transactions in the Bitcoin network in **Sections 2.4.2** and **2.4.3**, respectively. In **Section 2.4.4**, we summarize the differences in the literature and the work in this thesis.

2.4.1 Measurement tools

Neudecker [113] built a tool (that is not publicly available) for monitoring different aspects of the Bitcoin network. Though this tool provides valuable information on general network properties, including end-to-end propagation delays and churn, it does not provide detailed information about events related to the propagation of transactions and blocks at individual nodes, which is crucial for understanding the causes of delays and network inefficiencies.

Kalodner *et al.* [166] present *BlockSci*, a blockchain analysis tool designed to performs an analysis on transaction graphs, scripts, block indexes and other additional data which are view-able by the end-user. Though this tool is geared towards analysis of privacy and forensics, it lacks the ability to perform analysis on parameters such as propagation times of blocks, transactions missing from blocks, etc., which our logging system is able to achieve.

2.4.2 Block propagation and churn

Stutzbach and Rejaie [99] study churn in several peer-to-peer networks, specifically Gnutella, BitTorrent, and Kad. By inserting crawlers into each network, they characterize various metrics, such as peer inter-arrival time, session lengths, peer up time, peer down time etc., and fit distributions to the respective metrics. The authors state that "one of the most basic properties of churn is the session length distribution, which captures how long peers remain in the system each time they appear". Our work characterizes the statistics of session lengths and churn in the Bitcoin network, which to our knowledge have not been studied so far. Furthermore, our work is not limited to statistical characterization of churn, but also evaluates the impact of churn on the behavior of the Bitcoin network with respect to the efficacy of the compact block protocol.

Apostolaki *et al.* [111] simulate partitioning attacks on the Bitcoin network. In a partitioning attack, an attacker divides the network into multiple disjoint components, where no information flows between any two components. The authors incorporate churn in their simulations and assume that session lengths follow exponential distributions. We show in our work that, on an aggregate level, session lengths are better modeled by heavy-tailed distributions.

Decker and Wattenhofer [45] measure the time it takes for a block to propagate in the Bitcoin network. They show that the delay in propagation of blocks in the network results in forks in the Bitcoin blockchain. Since only one branch of the fork becomes part of the blockchain, nodes that create blocks in the other branch(es) essentially waste their power. Forks in the blockchain also lead to a phenomenon called *information eclipsing* which an adversary can leverage to perform a *double spending* attack. However, that work was published before the compact block protocol was implemented, *i.e.,* each block contained full transactions and no reconstruction was needed. Therefore, it does not capture the current behavior of block propagation, including additional delay incurred due to missing transactions in a compact block received by a node. We show in this work that churn can increase propagation delays of compact blocks received by a node in the Bitcoin network.

Neudecker *et al.* [110] study churn in the Bitcoin network from an attacker's perspective. They vary the session length of an attacking node in the Bitcoin network and, through simulations, show that a network partitioning "attack is sensitive to churn of the attacking node." However, they do not characterize churn in the network, and thus, it is unclear what is the basis for the parameters used in the simulations.

Karame *et al.* [112] study the security of using Bitcoin in fast payments, such as paying for a meal at a fast-food restaurant. They theorize that because of churn in the Bitcoin network, the connectivity of a victim node with the rest of the network varies with time. This gives an adversarial node considerable opportunities to connect with a victim node and perform a double spend attack. However, the authors neither characterize churn nor take it into account when performing analysis, measurements and experiments.

Augustine *et al.* [167] and Jacobs *et al.* [168] propose algorithms for efficient search and retrieval of data items in churn-tolerant peer-to-peer networks. These algorithms can help churning nodes retrieve transactions that they missed while being disconnected from the network. The algorithms assume that a churning node knows *a-priori* the ID of a peer that has the required data. Such an assumption does not hold in Bitcoin because Bitcoin is an unstructured peer-to-peer network [1]. Specifically, a node in Bitcoin does not know in advance which peer stores the data that it needs, and thus it broadcasts data requests to multiple peers [169].

Mišić *et al.* [170] study the improvements brought upon by the compact block relay protocol on the Bitcoin network. The queuing analysis presented by the authors shows that while the compact block relay protocol improves delivery times of blocks by up to 20% and reduces the probability of forks occurring in the network by up to 25%, it requires high transaction traffic to successfully recover transactions from their hashes in the compact blocks. However, the authors do not account for the presence of churn in the Bitcoin network when evaluating the performance of the compact block relay protocol.

Motlagh *et al.* [171–174] present an analytical model for the churning process in the Bitcoin network using continuous time Markov chain. The authors point out that churning nodes in the Bitcoin network not only affect the propagation of blocks in the network, but also consume network resources to synchronize their local copy of the blockchain with the rest of the network upon rejoining it. While these works complement our findings in this thesis, the authors present results from simulations based on an assumption that all churning nodes have the same session lengths. We present results based on data from the live Bitcoin network where session lengths of churning nodes are sampled from the actual distribution of up and down time of nodes in the network.

Ozisik *et al.* [123] propose the *Graphene* protocol, which couples an Invertible Bloom Lookup Table (IBLT) [156] with a Bloom filter in order to send transactions in a package smaller than a compact block. According to the authors, the size of a Graphene block can be a fifth of the size of a compact block, and they provide simulations demonstrating their system. This block propagation concept has recently been merged into the Bitcoin Unlimited blockchain. However, similar to the compact block protocol, Graphene assumes a large degree of synchronization between mempools of sending and receiving peers. In case of missing transactions, the receiving peer requests larger IBLTs from the sending peer, thus potentially adding significant propagation delay.

Mišić *et al.* [120] show that synchronizing mempools of churning nodes when they rejoin the network not only decreases the chances of missing transactions from compact blocks, but also reduces unnecessary network traffic when retrieving the aforementioned missing transactions from peers. While this work is complementary to our findings in this thesis, the authors present simulation-based findings whereas we report results from live Bitcoin nodes with an implementation of a synchronization protocol in the Bitcoin software.

Saad *et al.* [175] identify an inefficiency in the way Bitcoin Core relays blocks and transactions to peers which could potentially add several seconds of delay in their propagation to peers. The authors also show that the number of synchronized nodes that churn in the Bitcoin network in a 10-minute interval has almost doubled from 2019 to 2020 thereby reducing the synchronization of the network. However, the authors do not show how this affects the propagation of information in the Bitcoin network.

Naumenko *et al.* [176] propose the *Erlay* transaction dissemination protocol, the aim of which is to reduce the consumption of bandwidth due to dissemination of transactions across nodes in the Bitcoin network. The protocol uses *set sketches* to perform set reconciliation across mempools of nodes. The authors do not evaluate the protocol in the presence of churn, and it is unclear whether Erlay would perform efficiently when a node misses a large number of transactions from a block that it receives. We show in this work that a block received by a churning node can miss as many as 2,722 transactions.

Rohrer *et al.* [67] present *Kadcast*, a protocol which, unlike mainstream blockchain systems such as Bitcoin or Ethereum, uses a structured overlay network to propagate blocks in the blockchain networks. The authors show via simulations that their protocol performs better in terms of propagation delay of blocks than “VanillaCast”, a framework representative of most blockchain networks. To the best of our knowledge, however, the authors do not account for churn in their simulations. It is, therefore, not known how well the protocol performs in such conditions.

2.4.3 Orphan transactions

To the best of our knowledge, there is 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 [41] 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.* [40] 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 [177]. Both of these vulnerabilities were responsibly reported and fixed [42, 178]. 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.

2.4.4 Summary

In this section, we highlight the shortcomings in the related work surveyed in **Sections 2.4.1 to 2.4.3**.

- Existing tools that facilitate extraction of data from blockchains are either (i) not publicly available; or (ii) limited in scope of what they can measure. Therefore, they do not provide adequate support to study occurrence and impact of natural phenomena such as churn and orphan transactions in the blockchain systems.

- The works surveyed above (i) do not seek characterization of churn while incorporating it in the analyses presented thereby making the choice of relevant parameters unclear; (ii) evaluate the impact of churn on simulated data; and/or (iii) do not show the effects of churn on the performance of the Bitcoin network. The characteristics and impact of churn in the Bitcoin network thus remain unknown.

- Prior work uses orphan transactions in the Bitcoin network to infer the topology of the network. However, it does not seek a characterization of the orphan transactions or study their performance overhead. Furthermore, the relationship between the churn behavior of a full node and the transactions that it receives becoming orphan remains unknown.

In **Chapters 3** to **5**, we fill these gaps through application of empirical and statistical methods.

Chapter 3

Churn in the Bitcoin network

A key challenge for the Bitcoin network lies in reducing the propagation time of blocks. The *compact* block relay protocol [54] (described in greater detail in **Section 2.2.4**) was proposed to help address this challenge, and it is currently implemented on the standard Bitcoin Core reference implementation. This protocol aims to decrease block propagation time to the broader network by reducing the amount of data propagated between nodes. However, as with other peer-to-peer networks, it is also important that the Bitcoin network be able to support a high rate of *churn* [99], that being the rate at which nodes independently enter and leave the network. That is, the network should be able to quickly propagate blocks to all current nodes, even when some of these nodes frequently enter and leave the network. While churn has been extensively studied in different peer-to-peer networks [99–105], it has received little attention in relation to blockchains [49, 106–112]. As a result, questions remain about the extent of churn in the Bitcoin network and its effect on the relay of blocks.

We answer the abovementioned questions in this chapter based on our work published in the proceedings of the *IEEE International Conference on Blockchain and Cryptocurrency 2019* [68], and the *IEEE Transactions on Network and Service Management* [69]. The rest of the chapter is organized as follows. In **Section 3.1**, we describe our methodology for obtaining and processing data on churn in the Bitcoin network, and conduct a statistical analysis of the data. In **Section 3.2**, we detail the experimental setup for evaluating the impact of churn on block relay, and present the results. In **Section 3.3**, we introduce the `MempoolSync` protocol and report experimental results on the efficiency of this synchronization protocol. We present a discussion on and limitations of our work in **Section 3.4**. Finally, **Section 3.5** summarizes the main findings in this chapter.

3.1 Churn characterization

Nodes on the Bitcoin network may leave and rejoin the network independently. As a result, characterizing churn requires observation of the node activity on the network. In this section, we first detail our methodology to obtain and process data on churn, and then we present our statistical analyses. In **Section 3.2**, we leverage this characterization to run experiments on the compact block protocol with churning nodes.

3.1.1 Obtaining and processing data

The site Bitnodes [179] continuously crawls the Bitcoin network and provides a list of all reachable nodes with an approximately 5-minute resolution. Each network snapshot is available for roughly 60 days [180], and the website provides a rich API interface that can be used to download each snapshot as a JavaScript Object Notation (JSON) file containing the IP address, version of Bitcoin running, geographic location etc., of the nodes on the network that are up. **Listing 3.1** shows an example of a JSON snapshot taken by the crawler at the UNIX timestamp 1526742217.

Our analysis was based on all available JSON files from Saturday, May 19, 2018 11:03:37 AM EST (UNIX timestamp: 1526742217) to Tuesday, July 17, 2018 04:06:14 PM EST (UNIX timestamp: 1531857974) including a total of 14,674 snapshots ordered according to unique UNIX timestamps.

We parsed each JSON file and generated a dataset of all IP addresses that appear at any point on the Bitcoin network during the aforementioned time period, totaling

```
"220.75.229.130:3927": [
    70015,                 Protocol Version
    "/Satoshi:0.13.2/",  User Agent
    1526337217,            Connected Since
    13,                    Services
    165277,                Height
    "220.75.229.130",      Hostname
    "Seoul",               City
    "KR",                  Country Code
    37.5985,               Latitude
    126.9783,              Longitude
    "Asia/Seoul",          Timezone
    "AS4766",              ASN
    "Korea Telecom"        Organization Name
]
```

Listing 3.1: Part of a JSON file transmitted to a Bitcoin node.

4.77×10^4 distinct IP addresses. We find that out of these IP addresses, 4.65×10^4 ($> 97.5\%$) announce that they have the `NODE_NETWORK` [181, 182] service enabled, *i.e.,* they are able to provide a copy of the full blockchain. Out of the about 2% remaining nodes, a large fraction ($> 60\%$) is running in pruned mode, *i.e.,* they are able to provide at least the last 288 blocks. Therefore, it is evident that almost all nodes in our data set are comprised of full nodes that partake in dissemination of information in the Bitcoin network.

Given the list of IP addresses, we then ran a script that looks for each IP address through each consecutive network snapshot. If an IP address was found in two consecutive network snapshots, we concluded that the IP address was *up*, and thus online, for 10 minutes (recall Bitnodes' 5-minute resolution). Similarly, if the IP address was found in only one of the two consecutive network snapshots, we inferred that the node either left or rejoined the network. Finally, if IP addresses that were not found in any of the two consecutive network snapshots were designated as *down* (*i.e.,* offline) for 10 minutes. This allowed us to record the networked behavior of a node, *i.e.,* the duration of time it is on and off the Bitcoin network over the 1.47×10^4 snapshots.

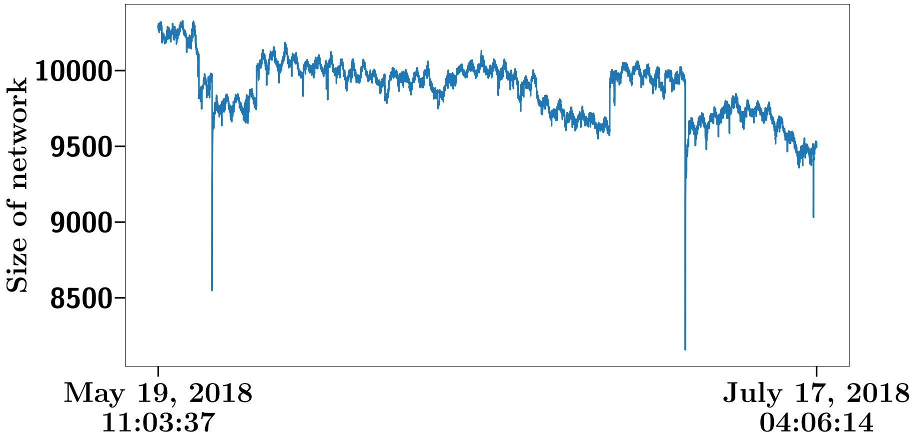


Figure 3·1: Size of Bitcoin network over the measurement period.

3.1.2 Churn rate

Out of 4.77×10^4 distinct IP addresses observed on the network during the aforementioned time period, only 1.15×10^3 (*i.e.,* 2.42% of the nodes) appeared to be online at all times. Nodes corresponding to the remaining IP addresses contributed to churn in the Bitcoin network.

Prior work [99, 183] showed that the overall size of peer-to-peer networks (*i.e.,* the total number of peers) stays relatively stable over time. Indeed, **Figure 3·1** depicts the number of reachable nodes in the Bitcoin network extracted from successive snapshots, where, on average, there are 9,881 reachable nodes with a standard deviation of 186. The low deviation, in addition to visual inspection of **Figure 3·1**, suggests that the size of the Bitcoin network is indeed stable over the measurement period.

Next, we evaluated the *churn rate*, namely the rate at which nodes oscillate between up and down sessions. More precisely, the churn rate can be defined as $R = 1/T$, where T is a random variable corresponding to the sum of the duration of an up session and its subsequent down session. **Figure 3·2** shows the complementary cumulative distribution function (CCDF) of the churn rate R as measured across all the observed Bitcoin nodes. Note that R exceeds one churn per node per day, with probability greater than 45%, and that there is a 10% probability of $R \geq 9$ churns

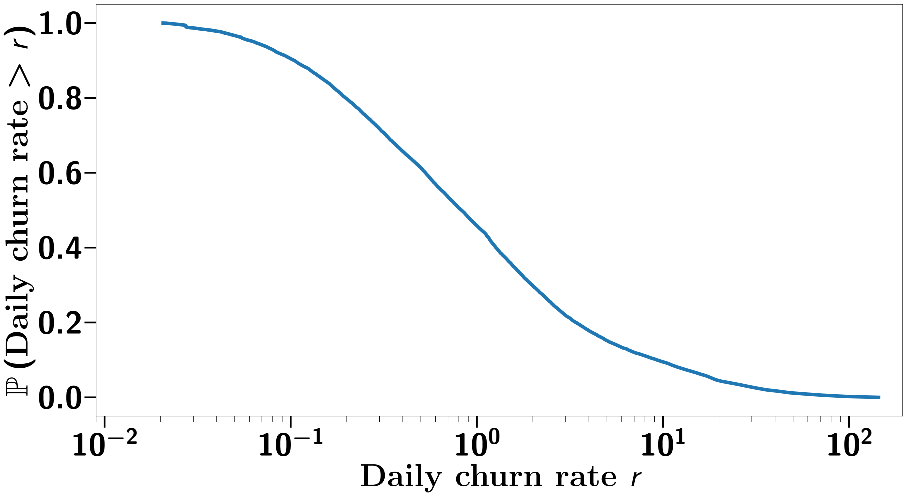


Figure 3·2: Daily churn rate on the Bitcoin network.

per node per day. The average churn rate per node is $\bar{R} = 4.16$ per day.

We point out that the nodes that are always up do not contribute to churn in the Bitcoin network. For this reason, we filtered out data related to these nodes from our data sets on the session lengths of a node's up and down time on the network. In addition, we filtered out the first and the last session (whether up or down) of each node, because we did not know how long a node was up or down before we started and after we finished our measurement.

3.1.3 Statistical fitting of session lengths

Prior work [184–187] showed that session lengths in various peer-to-peer protocols exhibit a behavior similar to a heavy-tailed distribution. Therefore, in our statistical fitting, we focus on fitting heavy-tailed distributions to the data, specifically: the *generalized Pareto distribution*, *the log-normal distribution*, the *Weibull distribution* [188] and the *log-logistic distribution*. Nolan [189] shows that maximum likelihood estimation (MLE) of heavy-tailed distribution parameters is feasible. Hence, we use MATLAB's distribution fitting capabilities [190] to fit distributions based on the MLE criterion. Finally, we also consider the exponential distribution, as a basis for comparison.

Up sessions. We first fit a distribution to the data representing up session lengths. Our fitting applies to the first 25,000 minutes, which roughly translates to 2.5 weeks. We used the following criteria to determine the goodness-of-fit of the various distributions to the actual data:

1. The R-squared value given by

$$\mathrm{R}^2 = 1 - \frac{\sum_{i=1}^{n} (y_i - \hat{y}_i)^2}{\sum_{i=1}^{n} (y_i - \bar{y})},$$

   where y is the actual value, $\hat{y}$ is the calculated value, and $\bar{y}$ is the mean of y [191]. An $\mathrm{R}^2 = 1$ suggests a perfect model [192].

2. The root mean squared error (RMSE) given by

$$\mathrm{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n} (\hat{y}_i - y_i)^2},$$

   where y is the actual value and $\hat{y}$ is the calculated value. [191] An RMSE $= 0$ indicates that all of the calculated values lie on the line formed by the actual values [193].

3. Visual inspection of the data.

We set the parameter values generated by MATLAB as a base and performed an exhaustive search within ±10% of the base parameters. The final results for each distribution are the highest R^2 and lowest RMSE in that range.

The results can be seen in **Figure 3·3**, and the R^2 and RMSE scores for the fits are detailed in **Table 3.1**. A key observation is that the *exponential distribution* is a very poor fit for the session lengths. While the *log-normal distribution* performs

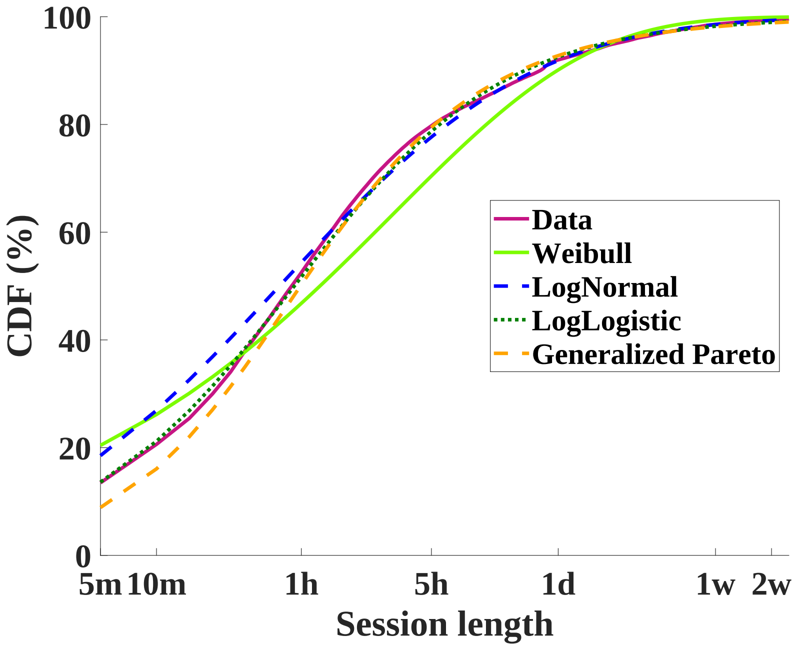


Figure 3·3: Distribution fits for "up session" lengths.

Distribution	R^2	RMSE
Weibull	0.9002	2.60e−03
Log-normal	0.9939	1.29e−06
Log-logistic	0.9907	1.80e−03
Generalized Pareto	0.9856	2.20e−03
Exponential	0.4904	21.70e−03

Table 3.1: R^2 and RMSE scores of distribution fits for "up session" lengths.

the best in terms of R^2 and RMSE scores, **Figure** 3·3 indicates that the *log-logistic distribution* fits best the cumulative distribution function (CDF) of the empirical data, at least in the initial portion where most of the data lies. Therefore, judging from the combination of **Figure** 3·3 and **Table** 3.1, we conclude that the *log-logistic distribution*, given by

$$F_{(\alpha,\beta)}\left(x\right) = \frac{1}{1 + \left(x/\alpha\right)^{-\beta}}, \tag{3.1}$$

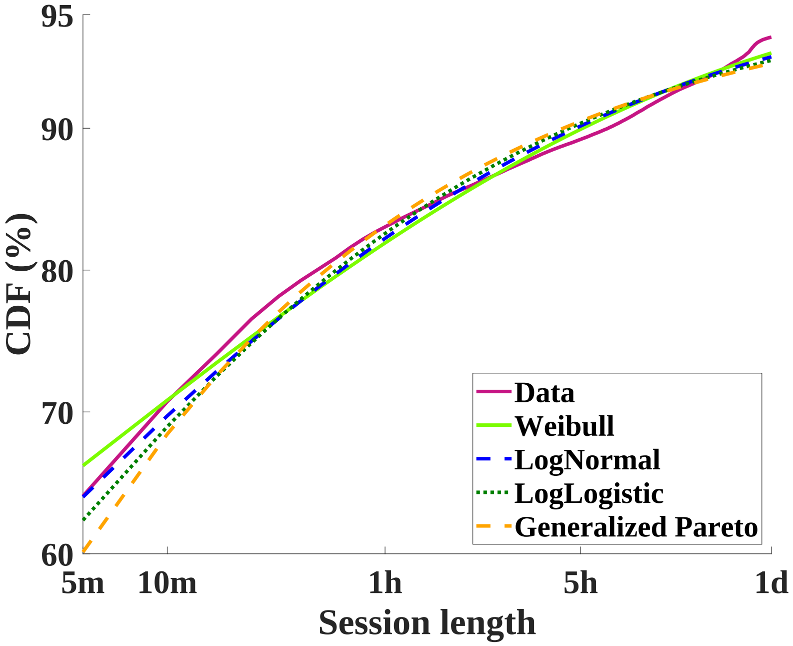


Figure 3·4: Distribution fits for "down session" lengths.

where $\alpha > 0$ is the scale parameter, and $\beta > 0$ is the shape parameter, is the best fit for the up sessions. The parameters for the *log-logistic distribution* fit in **Figure 3·3** are $\alpha = 11.000$ and $\beta = 0.771$.

Down sessions. Next, we fit distributions to the data representing down session lengths. We employed an approach similar to that in the previous section. We focused on performing a statistical fitting for sessions that are down for up to one day (representing over 93% of the cases). Note that the mempool of a node that is continuously off the network for a duration exceeding one day will largely be out of synchronization with the rest of the network.

The fitting results are shown in **Figure 3·4**. The corresponding R^2 and RMSE scores are listed in **Table 3.2**. Notice that the *exponential distribution* is a very poor fit and is never able to achieve an R^2 value above 0. Observing the combination of **Figure 3·4** and **Table 3.2**, we conclude that in this case the *Weibull distribution*,

Distribution	**R^2**	**RMSE**
Weibull	0.9777	3.28e−04
Log-normal	0.9694	5.36e−04
Log-logistic	0.9575	7.72e−04
Generalized Pareto	0.9429	9.55e−04
Exponential	0	1

Table 3.2: R^2 and RMSE scores of distribution fits for "down session" lengths.

given by

$$F_{(\lambda,k)}(x) = \begin{cases} 1 - \mathrm{e}^{-(x/\lambda)^k} & x \geq 0 \\ 0 & x < 0, \end{cases} \tag{3.2}$$

where $\lambda > 0$ is the scale parameter, and $k > 0$ is the shape parameter, is the best fit for the down sessions. The parameters for the *Weibull distribution* fit in **Figure** 3·4 are $\lambda = 0.640$ and $k = 0.183$.

Table 3.1 and **Table** 3.2 show that the exponential distribution is not a suitable fit for either the `up` session lengths or the `down` session lengths. This suggests that on an aggregate level, Markov process may not be suitable for performing analysis on churn in the Bitcoin network. Instead, we believe that churn should be analyzed using alternating renewal processes with heavy-tailed session lengths [194].

3.1.4 Subnet analysis

In this section, we investigate churn behavior at the level of IP subnetworks (subnets). Our dataset contains 3.96×10^4 IPv4 nodes and 7.51×10^3 IPv6 nodes; a tiny fraction of the remaining nodes uses onion routing [195]. We first focus our analysis on IPv4 /24 subnets, identifying 29,036 such subnets over 99% of which contain fewer than 10 Bitcoin nodes (*i.e.,* with unique IP addresses). The average number of Bitcoin nodes

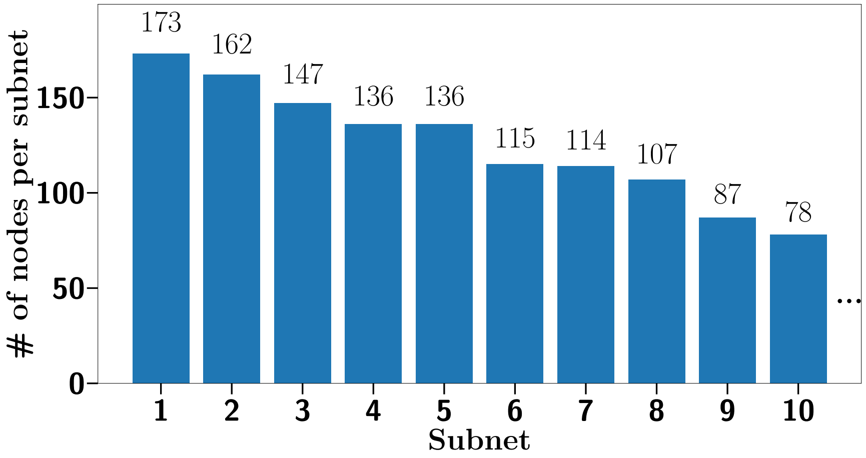


Figure 3·5: Largest IPv4 /24 subnets sorted in descending order.

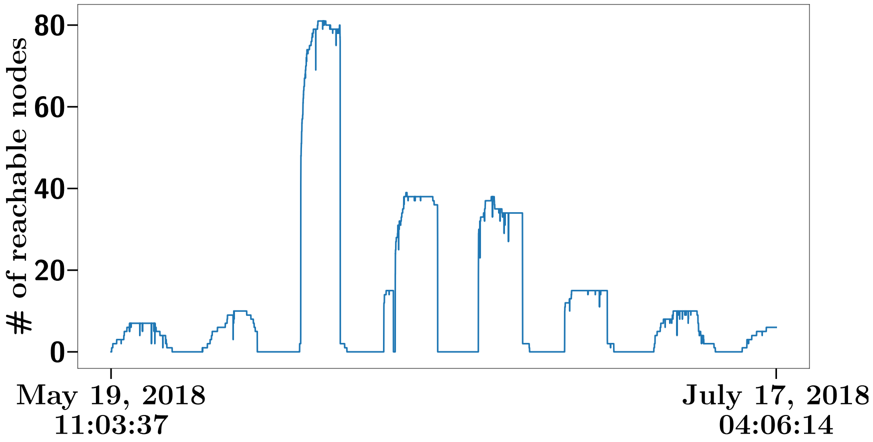


Figure 3·6: Number of reachable nodes in the largest IPv4 /24 subnet in consecutive Bitcoin network snapshots.

per subnet is 1.36. **Figure** 3·5 shows statistics for the 10 largest subnets, falling quickly from a maximum of 173 nodes to below 100.

Figure 3·6 depicts the evolution of the number of reachable nodes over time in the largest subnet. A prominent pattern emerges: all the nodes in the subnet are periodically unreachable for roughly the same time duration. Another interesting insight is that the 173 nodes recorded in this subnet do not appear on the Bitcoin network at the same time. In fact, at most 81 unique nodes are reachable at the same time. We observe similar behavior in the next nine largest subnets.

We next study the *duty cycle*, defined as the fraction of time during which a node is reachable on the Bitcoin network. **Figure 3·7** shows the CDF of the duty cycle of nodes belonging to the largest subnet. The highest duty cycle of a node in this subnet is 0.25. On average, a node in this subnet has a duty cycle of 0.07 with a standard deviation of 0.04. **Figure 3·8** shows the CDF of the duty cycle in the next nine largest subnets, which is very similar to that shown in **Figure 3·7**.

The similarity between **Figure 3·7** and **Figure 3·8** raises an important question: is churn behavior in the 10 largest subnets correlated? We consider the 10 largest subnets and compute the correlation of churn for nodes within the same subnet and in different subnets. We use the Pearson correlation coefficient [196] to measure correlation between the presence of nodes on the Bitcoin network. Given two data sets D_1, D_2, the Pearson correlation coefficient, ρ, where $-1 \leq \rho \leq 1$, is given by

$$\rho_{D_1,D_2} = \frac{\operatorname{cov}(D_1, D_2)}{\sigma(D_1)\,\sigma(D_2)},$$

where $\operatorname{cov}(D_1, D_2)$ represents the covariance between the two data sets, D_1 and D_2, $\sigma(D_1)$ represents the standard deviation of the data set D_1, and $\sigma(D_2)$ represents the standard deviation of the data set D_2 [197]. **Figure 3·9** shows the results in the form of a correlation matrix. While the behavior of nodes within the same subnet may show correlation with one another, the behavior of nodes across subnets is largely uncorrelated. This finding indicates that across the 10 largest IPv4 /24 subnets nodes independently contribute to churn in the Bitcoin network.

3.1.5 Geographic analyses

Figure 3·10 shows the geographical locations (based on data obtained from Bitnodes) of the 47,702 individual nodes discovered in the Bitcoin network during the time period mentioned in **Section 3.1.1**. Nodes that are always up during this time period are marked white to make them distinguishable from the remaining nodes that

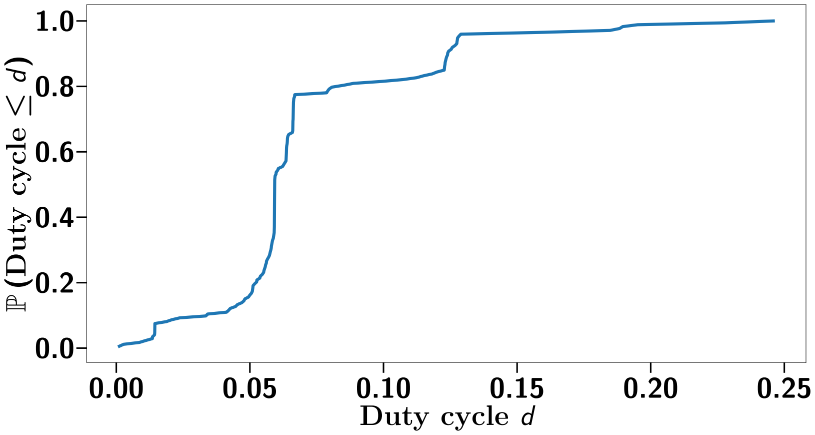


Figure 3·7: CDF of duty cycle of nodes in the largest IPv4 /24 subnet. The duty cycle of a node represents the fraction of a time it is reachable during the measurement period.

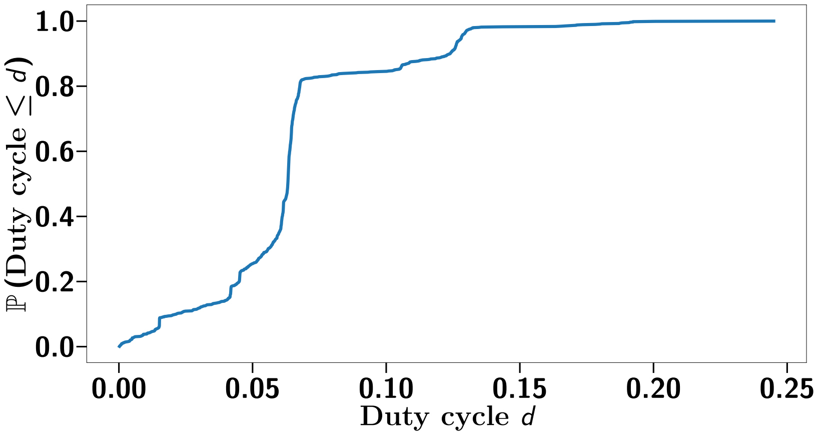


Figure 3·8: CDF of duty cycle of nodes in the 2^{nd} to 10^{th} largest IPv4 /24 subnets.

contribute to churn in the network. We observe that the majority of the Bitcoin nodes are located in the North America and Europe. South America, North Asia, the Far East and Oceania show a moderate presence while Africa and Central and South Asia show a very little presence of Bitcoin nodes. We note that the nodes that are always connected are not co-located in one region but rather spread out over the entire world map.

Our geographic distribution is summarized in **Table 3.3**, which shows that the North American and European continents have the highest percentage of nodes that

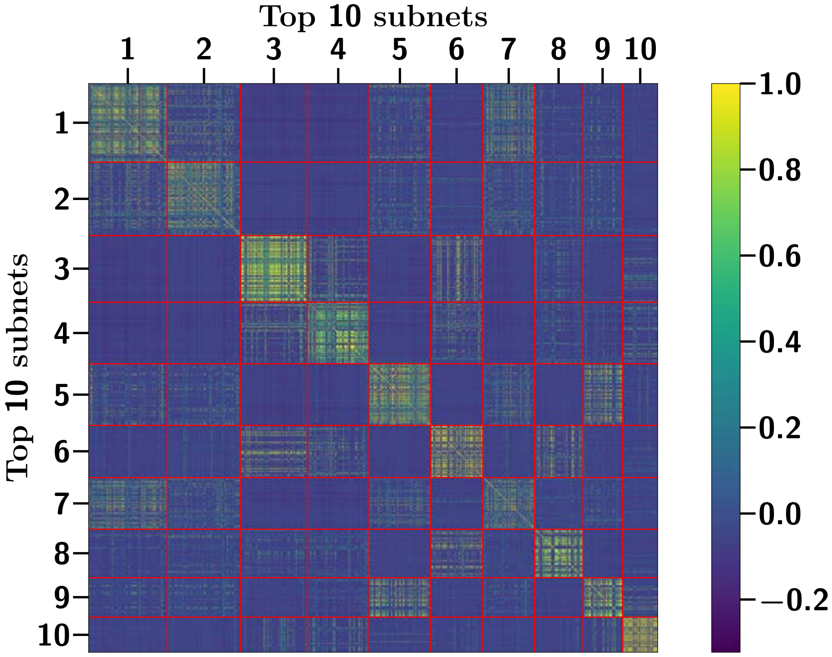


Figure 3·9: Correlation matrix showing the correlation between churn behavior of nodes in the 10 largest /24 IPv4 subnets. The red line delimiters separate between different subnets.

Continent	**Percentage**
Africa	0.051
Asia	1.138
Europe	3.239
Oceania	0.692
North America	3.414
South America	0.289

Table 3.3: Percentage of continuously connected nodes in each continent.

are always up. On the other hand, Africa has a very small percentage of nodes that are always up. These results may be related to the intermittent nature of Internet access in parts of that continent.

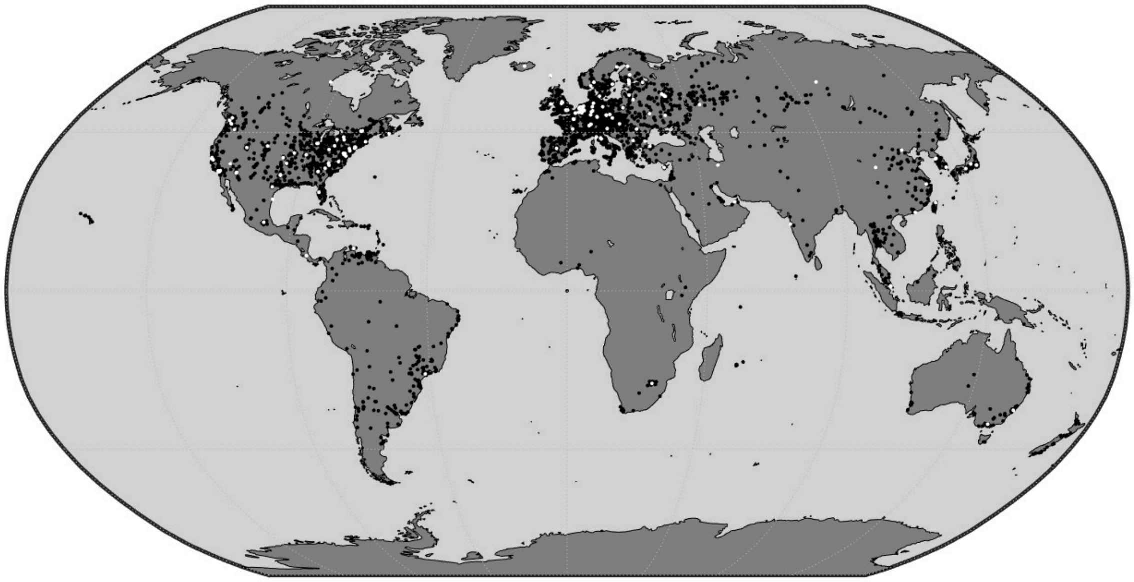


Figure 3·10: Geographic location of individual nodes on the Bitcoin network. Nodes that are always up are marked white. Remaining (black) nodes contribute to churn in the network.

3.2 Experimental analysis of compact block performance with churn

In this section, we evaluate the performance of block relay, and especially the compact block relay protocol, in the presence of churning nodes, to realistically reflect the behavior of the Bitcoin P2P network. The section details the mechanism that we developed to log events on the Bitcoin network, the experimental setup, the method for emulating churning nodes based on the distribution fits performed in **Section 3.1.3**, and finally, the results obtained.

3.2.1 Data collection mechanism

To aid in understanding Bitcoin Core's behavior, we have developed a new log-to-file system that produces human-friendly, easy-to-read text files. This logging system is open-source and we have made it available to the research community

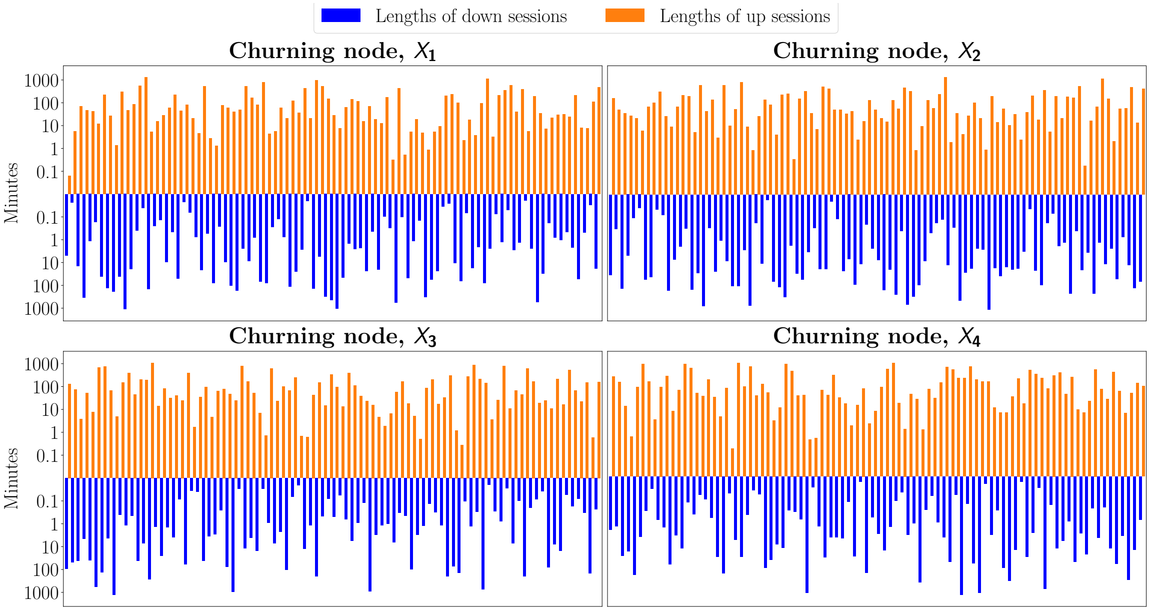


Figure 3·11: Sampled up and down session lengths.

(`[121]/src/logFile.*`). This new logging mechanism allows one to isolate specific behaviors through select calls anywhere within the Bitcoin Core's source code, most notably information about different protocols such as the compact block. The logging system writes core data to a log file, and also can record various events and the information associated with those events. For instance, when a compact block arrives, the system logs this event and saves the transaction hashes included in the compact block in a separate file with a unique identifier tying it to a log entry. We have used this system as our primary data collection mechanism for all of our experiments. We explain the system in more detail in **Appendix A** along with examples of usage and the corresponding logs generated.

3.2.2 Experimental setup

The aim of the experiment is to determine the efficiency of the compact block protocol in the presence of churn. We achieve this by running eight nodes in the Bitcoin network. The nodes are Dell Inspiron 3670 desktops, each equipped with an 8th

Generation Intel® Core i5−8400 processor (9 MB cache, up to 4.0 GHz), 1 TB HDD and 12 GB RAM. The nodes are each running the Linux Ubuntu 18.04.1 LTS distribution.

We ran experiments over a period of two weeks. Four nodes (denoted by X_1, X_2, X_3, X_4) used sampled session lengths to emulate churn on the network. Specifically, we generated samples of the best fit distributions given by **Equations (3.1)** and **(3.2)**, such that the aggregate sum of the up and down sessions is at least two weeks for each node. We limited both the up and down session lengths from a minimum of 1 second to a maximum of 1 day making sure that the mean of these session lengths is within 1% of the mean of the original session lengths used to characterize churn. The remaining four nodes (denoted by C_1, C_2, C_3, C_4) acted as *control nodes* that are continuously connected to the network. **Figure 3·11** shows the sampled up and down session lengths for each churning node used in the experiments. It is clear from the figure that each churning node emulates up and down sessions independent from other churning nodes. In order to avoid any bias, we used the Bitcoin RPC API `setban` [198, 199] to ensure that the eight nodes are not connected to each other as peers in the Bitcoin network. This way, our nodes did not directly influence each other. Our experiment started on Wednesday, May 27, 2020 12:00:00 EST and ran without interruption for two weeks. We have made all experimental logs publicly available on GitHub [122].

3.2.3 Statistics on compact blocks

We compare the number of compact blocks that the churning nodes (denoted by X_1, X_2, X_3, X_4) and the (stable) control nodes (denoted by C_1, C_2, C_3, C_4) fail to reconstruct. **Table 3.4** shows the results. The churning nodes are unable to reconstruct a larger fraction of compact blocks that they received as compared to the control nodes. Indeed, of the blocks they receive, the control nodes are able to recon-

Nodes	**Blocks Received**	**Successful Compact Blocks (%)**	**Unsuccessful Compact Blocks (%)**
C_1	1726	93.97	6.03
C_2	1453	91.53	8.47
C_3	1751	91.55	8.45
C_4	1899	94.05	5.95
X_1	1299	66.20	33.80
X_2	1278	73.08	26.92
X_3	1279	62.16	37.84
X_4	1198	66.03	33.97

Table 3.4: Block reception statistics for control nodes C_1, C_2, C_3, and C_4, and churning nodes X_1, X_2, X_3 and X_4.

struct successfully on average 1.59×10^3 blocks out of 1.71×10^3 blocks (*i.e.,* 92.85% of the blocks), while the churning nodes are able to reconstruct successfully on average only 8.45×10^2 blocks out of 1.26×10^3 blocks (*i.e.,* 66.88% of the blocks). The results are quite consistent across both the control and churning nodes.

3.2.4 Statistics on missing transactions

Churning nodes are generally missing far more transactions in blocks they are unable to reconstruct than the control nodes. We identify transactions missing from blocks by recording the requests for missing transactions that a node makes, upon receiving a new block. This is done using the log-to-file system described earlier (cf. Section IV-A).

We find that on average a churning node misses 78.08 transactions from a block with a standard deviation of 288.04 transactions, whereas a control node misses on average 0.87 transactions with a standard deviation of 10.78 transactions. **Figure 3·12** shows the CCDF of the number of missing transactions. From the figure, we observe that churning nodes may be missing up to thousands of transactions from a block they receive, while control nodes may miss at most a few hundred transactions. Roughly

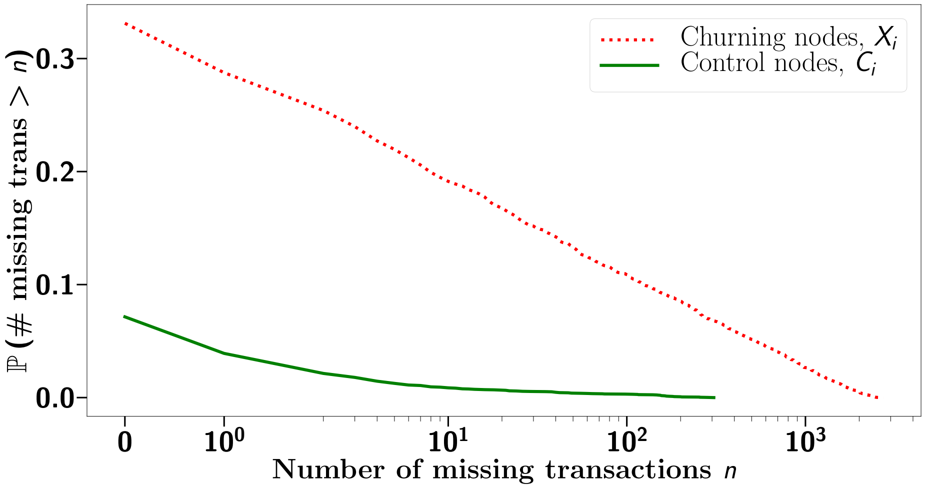


Figure 3·12: CCDF of number of missing transactions in churning and control nodes.

11% of blocks received by churning nodes miss more than 100 transactions up to as many as 2,722 missing transactions in a block. On the other hand, only about 0.3% of blocks received by control nodes miss more than 100 transactions up to a maximum of only 307 missing transactions in a block. Therefore, our results clearly indicate that churning nodes need to request a high number of transactions from their peers to successfully reconstruct a block.

3.2.5 Statistics on propagation delay

Next, we investigate whether and how transactions missing in a block delay the block's propagation. We measure propagation delay as the difference between the time at which a measurement node receives an announcement of a block, *i.e.*, an `inv` message with the hash of the block, from one of its peers and the time at which the node is able to successfully collect all missing transactions that are included in the block.

We compare the propagation delay of blocks received by churning nodes with the propagation delay of blocks received by control nodes. Blocks received by the control nodes experience an average propagation delay of 109.31 ms with a standard deviation

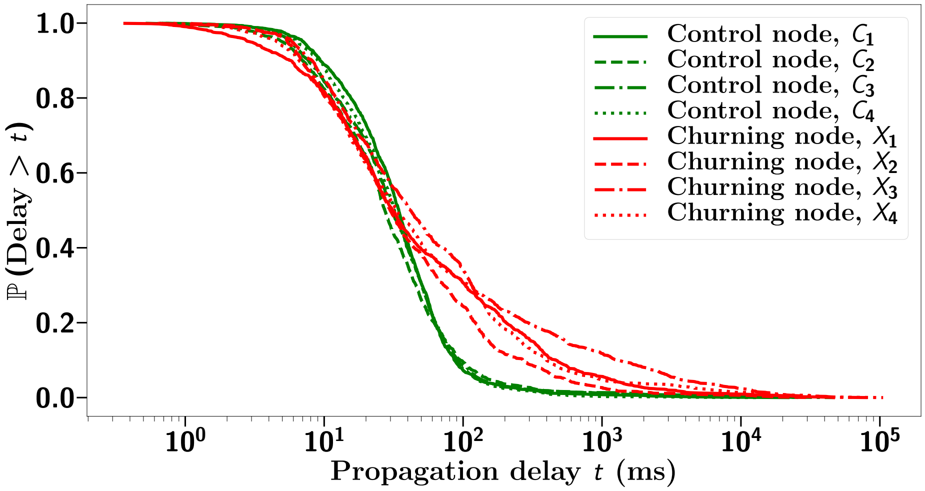


Figure 3·13: Propagation delay across all blocks for both churning and control nodes.

of 1,066.15 ms. Blocks received by the churning nodes, on the other hand, experience an average propagation delay of 566.89 ms with a standard deviation of 3,524.78 ms.

Figure 3·13 shows the CCDF of propagation delays of blocks received by all nodes. From the figure, we observe that blocks received by control nodes rarely have large propagation delays. On an aggregate level, only about 7% of blocks received by control nodes have a propagation delay larger than 100 ms with a maximum block propagation delay of 46.14 s. By comparison, on an aggregate level, roughly 30% of blocks received by churning nodes experience a propagation delay larger than 100 ms with a maximum propagation delay 105.54 s, more than twice that of any block received by control nodes.

3.3 MempoolSync

The experimental results from **Section 3.2** make it clear that missing transactions add significant delay to the propagation of blocks. This problem is especially acute for churning nodes since their mempools often miss transactions. To address this issue, we propose, implement and evaluate a new protocol to keep the mempools of churning nodes synchronized with the rest of the network. We call this protocol `MempoolSync`. The main goal of `MempoolSync` is to reduce the number of missing transactions in mempools and, consequently, the propagation delay of blocks. Note that `MempoolSync` does not attempt to minimize communication complexity, a well-known problem in the distributed computing literature [156, 158, 200] whose implementation we leave for future work. Rather, our implementation of `MempoolSync` into the Bitcoin Core demonstrates the key benefits of synchronizing the mempools of churning nodes with the rest of the network.

3.3.1 Design of MempoolSync

`MempoolSync` is designed to periodically synchronize the mempool of a churning node (*receiver*) with the mempool of a non-churning node (*sender*). **Figure 3·14** shows an overview of the synchronization protocol. The protocol leverages Bitcoin's existing functionality to package and send inventory (`inv`) messages, as well as request and propagate transactions. The sender selects transaction hashes from its mempool and packages them in a message (*inv*). The sender then sends the message to the receiver who, upon receiving the message, computes which of the hashes in the message are not present in its mempool. The receiver then requests the respective transactions from the sender (`getdata`), who in turn responds with the requested transactions (`tx`). Note that `MempoolSync`, does not require additional steps to ensure a receiving node actually receives all transactions that it requested. Instead, the protocol relies on the default Bitcoin behavior to send transactions. The sender then waits for a configurable amount of time before repeating the process.

`MempoolSync` is a one-way synchronization protocol, *i.e.,* the sender has no prior knowledge of the state of the receiver's mempool. Hence, an important question arises here: which transaction hashes should the sender select in each synchronization round? Our solution is based on the reference implementation of the algorithm

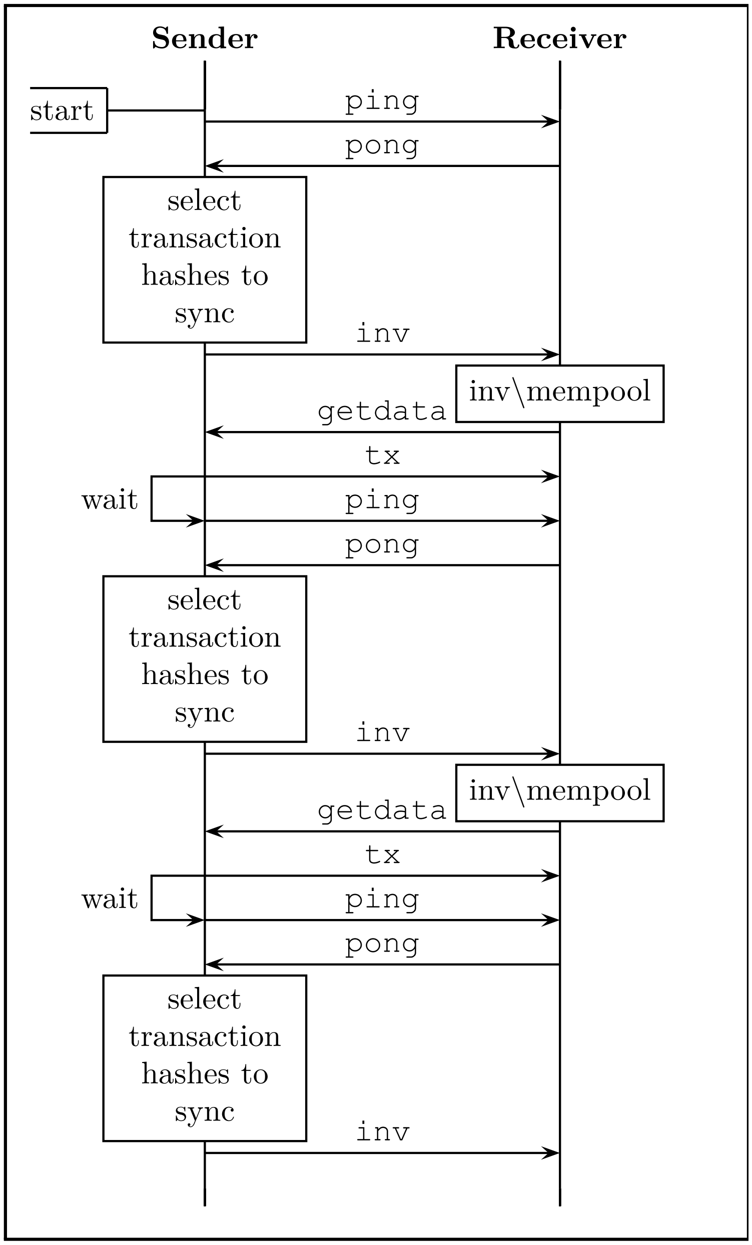


Figure 3·14: Exchange of messages between the non-churning node (*sender*) and the churning node (*receiver*) in the `MempoolSync` protocol.

for miners [201] in the Bitcoin Core. This reference suggests that miners should prioritize transactions based on their `ancestor_score` [202]. The `ancestor_score` is an internal Bitcoin scoring mechanism that ranks a transaction according to the total

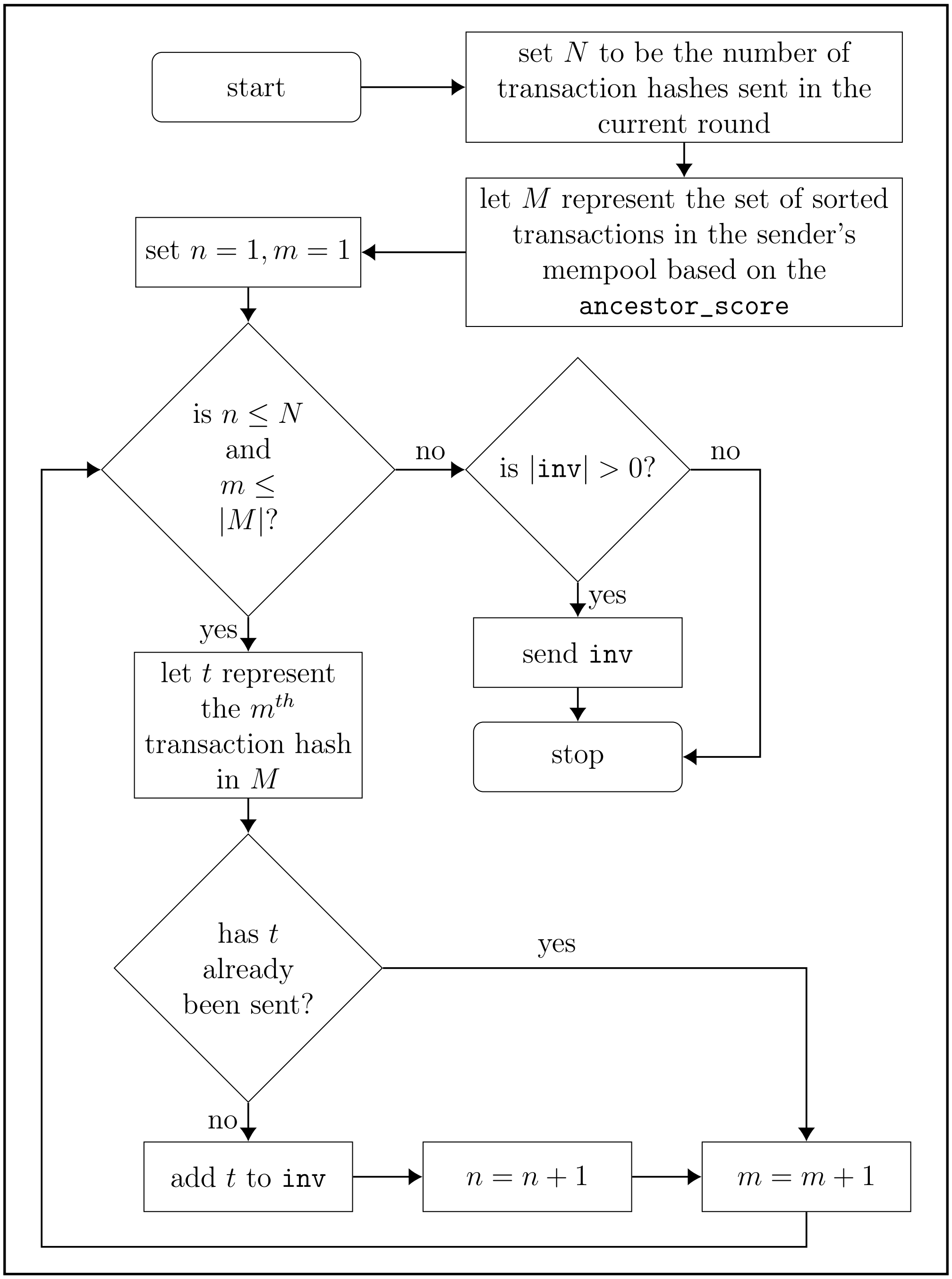

start
set N to be the number of transaction hashes sent in the current round
let M represent the set of sorted transactions in the sender's mempool based on the ancestor_score
set $n = 1, m = 1$
is $n \leq N$ and $m \leq |M|$?
no
is |inv| > 0?
no
yes
send inv
stop
yes
let t represent the m^{th} transaction hash in M
has t already been sent?
yes
no
add t to inv
$n = n + 1$
$m = m + 1$

Figure 3·15: Procedure for selecting transaction hashes to be included in the `inv` message in each round.

unconfirmed transaction fees in its ancestor tree. The sender in the `MempoolSync` protocol mimics this prioritization and likewise selects transaction hashes based on the respective transactions' `ancestor_score`. Indeed, one can expect that these transactions are the most likely to be included in upcoming blocks.

Figure 3·15 shows a general overview of how transaction hashes are selected and inserted in an `inv` message in each round of `MempoolSync`. For the sake of efficiency, the protocol ensures that the sender does not send the same transaction hash more than once. Indeed, the sender keeps track of the hashes it has previously sent, and omits re-sending them again in future rounds. The sender achieves this by storing hashes of transactions already sent to a peer in a `C++ std::map` [203] data structure. Hashes that are no longer in the mempool of the sender are periodically removed from the data structure to avoid memory overhead.

We next detail how the sender smartly decides the number of transaction hashes to include in an `MempoolSync inv` message. Denote by `N` the number of transactions that are packaged into the `MempoolSync inv` message. Next, denote the number of transactions in the sender's mempool by `NumTXsMP`, and the default number of transaction hashes to be sent in a single `MempoolSync inv` message by `DefTXtoSync`. Let `Y` represent a fraction of the mempool size (*i.e.,* a number between between 0 and 1).

By default, `MempoolSync inv` message should contain `DefTXtoSync` transaction hashes. That is,

$$\texttt{N} = \texttt{DefTXtoSync}.$$

However, the sender must take care of a couple of edge cases:

1. It is possible that the number of transactions in the sender's mempool far exceeds the default number of transaction hashes that the `inv` message should contain, *i.e.,* $\texttt{NumTXsMP} \gg \texttt{DefTXtoSync}$. When this happens, it makes sense to synchronize a larger fraction `Y` of transactions hashes from the sender's mem-

pool. That is,

$$\mathtt{N} = \max\left(\mathtt{DefTXtoSync}, \mathtt{Y} \times \mathtt{NumTXsMP}\right).$$

2. Similarly, it is possible that the sender does not have enough transactions in its mempool, *i.e.,* `NumTXsMP` < `DefTXtoSync`. This could happen when the sender has just joined the network, or it has just received a new block which causes removal of transactions from its mempool. When this happens, the sender synchronizes its *entire* mempool with the churning node. That is,

   $$\mathtt{N} = \min\left(\mathtt{DefTXtoSync}, \mathtt{NumTXsMP}\right).$$

   Ignoring this edge case would cause exceptions when running the Bitcoin software if the node tries to retrieve more transactions than available in the mempool.

We next provide a simple example to illustrate how transactions are chosen to be sent in a `MempoolSync` message. In this example, there are ten transactions in the sender's mempool, *i.e.,* `NumTXsMP` = 10. The protocol has smartly chosen the number of transactions to be included in the `MempoolSync` message to be five, *i.e.,* `N` = 5. **Table 3.5(a)** shows the hashes of transactions in the mempool of the sender along with their `ancestor_score` before they are sorted. **Table 3.5(b)** shows the same hashes sorted according to their `ancestor_score` in a descending order. The sender now has to pick five hashes from this sorted list of hashes. However, it also has to make sure it does not re-send any hashes that have already been sent to the receiver as shown in **Table 3.5(c)**. It can be seen that some of these hashes are in the top five positions in the sorted list of hashes. Therefore, while picking five transaction hashes from this list, the sender skips over any hashes that it finds in **Table 3.5(c)**, resulting in a list of hashes as shown in **Table 3.5(d)**. These hashes are packed into a `MempoolSync` message and sent to the receiver.

We have added an implementation of the `MempoolSync` protocol to a fork of the Bitcoin Core software as a proof-of-concept [121]. To make sure that the protocol does not interfere with, or worse, stall the main thread in the software, our implementation spawns a new thread when a Bitcoin client is started up. All operations related to the protocol strictly take place within this new thread.

Our implementation of the `MempoolSync` protocol relies on a connection manager maintained by each node. The connection manager, among other attributes, contains a list of addresses of peers to which the node is connected. In the `MempoolSync` protocol, all participating nodes are identified by their IP addresses. A node acting as sender transmits `MempoolSync inv` messages to peers listed in the connection manager. By default, when a peer disconnects from a Bitcoin node, the former remains in the latter's connection manager for up to 20 minutes even after it has disconnected [204]. Note that a sender always *pings* a receiver before sending to it the `MempoolSync inv` message containing transaction hashes. This way, the sender will not send `inv` messages to nodes that are down or unreachable.

3.3.2 Experimental evaluation of MempoolSync in the presence of churn

We performed an empirical evaluation of the `MempoolSync` protocol in the presence of churn. In this section, we describe our experimental setup and then present the results in the following sections.

We ran this experiment in parallel with the experiment in **Section 3.2.2** by adding four additional nodes (denoted by M_1, M_2, M_3, M_4) with similar hardware capabilities to the experimental setup. Nodes M_i, where $i \in \{1, 2, 3, 4\}$, emulated churn with up and down session lengths independently sampled from the distributions obtained in **Section 3.1.3** for each node. **Figure 3·16** illustrates the sampled session lengths. In addition, these nodes were also configured to accept `MempoolSync`

TX hash	Ancestor Score
`69dc6c`	586
`9d9816`	34
`ea844d`	440
`fa8082`	495
`a31fa4`	592
`824da7`	16
`4c09b6`	212
`d5a820`	474
`28d3b6`	833
`fa8ffc`	504

(a)

TX hash	Ancestor Score
`28d3b6`	833
`a31fa4`	592
`69dc6c`	586
`fa8ffc`	504
`fa8082`	495
`d5a820`	474
`ea844d`	440
`4c09b6`	212
`9d9816`	34
`824da7`	16

(b)

TX hashes		
`69dc6c`	`d5a820`	`4c09b6`

(c)

TX hashes				
`28d3b6`	`a31fa4`	`fa8ffc`	`fa8082`	`ea844d`

(d)

Table 3.5: An illustration of **(a)** *unsorted* transactions in the mempool with their ancestor scores (in *satoshis)*, **(b)** *sorted* transactions in the mempool with their ancestor scores (in *satoshis)*, **(c)** hashes of transactions already sent to a peer, and **(d)** transaction hashes sent in `MempoolSync` message when $N = 5$.

messages. Note that nodes can be configured as either senders or receivers in the `MempoolSync` protocol by setting the appropriate preprocessor macros to 1 as documented in the file `src/logFile.h` available in our GitHub repository [121].

The control nodes C_i from **Section 3.2.2** acted as the *sending* nodes in the `MempoolSync` protocol. Each churning node M_i was connected to a different sending node C_i. A preliminary measurement shows that a waiting time of 30 seconds in the `MempoolSync` protocol produces the best results. Therefore, we configured all sending nodes C_i to send a `MempoolSync` message after every 30 seconds. The parameter `Y` (which controls the size of the `MempoolSync` message as a fraction of the mempool size in each control node) was set to 0.1 and the parameter `DefTXtoSync` (which controls the default number of transactions sent in a single `inv` message) was set to 1,000. We found from a test measurement that some of the transactions sent as part

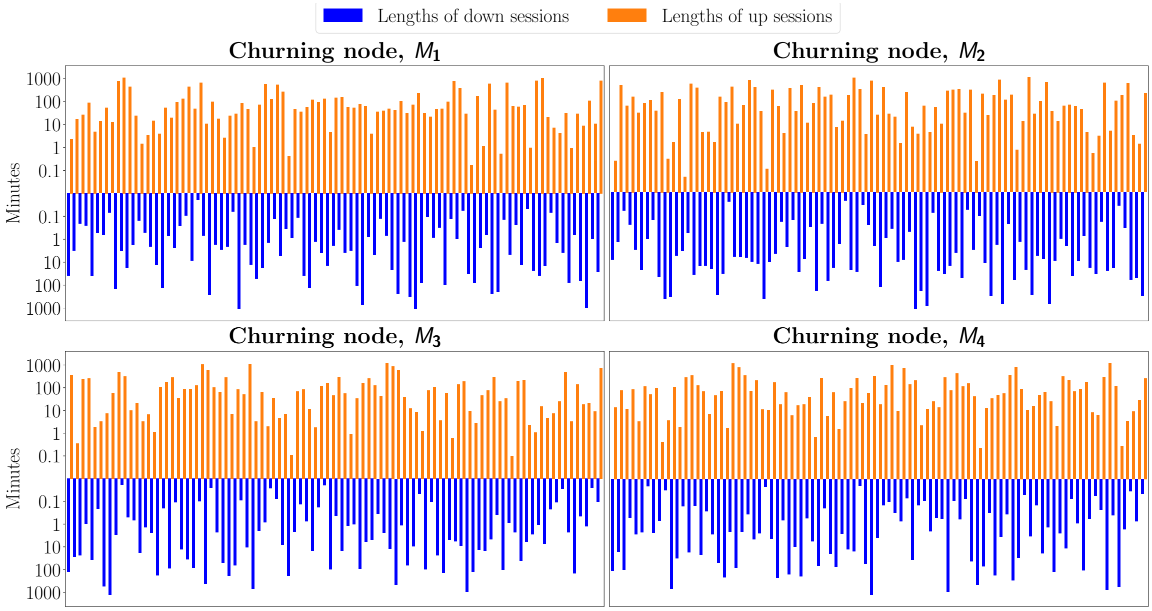


Figure 3·16: Sampled up and down session lengths.

of the `MempoolSync` protocol may end up as orphan. To make sure that `MempoolSync` does not cause unnecessary eviction of transactions already in the orphan pool, we increased the orphan pool size to 1,000 transactions. Prior work [39] shows that the chances of orphan transactions getting evicted with an orphan pool of this size are quite low.

To avoid biases, none of the nodes connected to one another as peers in the Bitcoin network (except obviously for the pairs (C_i, M_i)). The experiments ran without interruption from Wednesday, May 27, 2020 12:00:00 EST for two weeks. To avoid sending unnecessary traffic to other peers in the Bitcoin network, we made sure that each node C_i only sends `MempoolSync inv` messages to node M_i. We have made all experimental logs publicly available on GitHub [122].

Note that the statistics for the sending nodes are the same as nodes C_i in **Section 3.2**. Similarly, statistics for churning nodes that do not accept `MempoolSync` are the same as nodes X_i in **Section 3.2**. Therefore, in the following sections, we only compare the statistics between churning nodes that accept `MempoolSync` mes-

Nodes	**Blocks Received**	**Successful Compact Blocks (%)**	**Unsuccessful Compact Blocks (%)**
M_1	1142	80.82	19.18
M_2	1184	84.54	15.46
M_3	1247	80.91	19.09
M_4	1270	86.30	13.70
X_1	1299	66.20	33.80
X_2	1278	73.08	26.92
X_3	1279	62.16	37.84
X_4	1198	66.03	33.97

Table 3.6: Block reception statistics for churning nodes M_1, M_2, M_3, and M_4 that accept `MempoolSync` messages, and churning nodes X_1, X_2, X_3, and X_4 that do not accept such messages.

sages, *i.e.,* nodes M_i, and churning nodes that do not accept `MempoolSync` messages, *i.e.,* nodes X_i.

3.3.3 Statistics on compact blocks

Table 3.6 compares the percentage of successful compact blocks between the churning nodes M_i and X_i. The data in the table shows that churning nodes that accept `MempoolSync` messages always reconstruct a larger proportion of compact blocks that they received as compared to churning nodes that do not accept `MempoolSync` messages. The churning nodes X_i that do not implement `MempoolSync` successfully reconstruct, on average, only 66.88% of the compact blocks that they receive. By comparison, churning nodes M_i, that do implement `MempoolSync` successfully reconstruct, on average, more compact blocks *i.e.,* 83.19%. This finding indicates that `MempoolSync` leads to significant performance improvement.

3.3.4 Statistics on missing transactions

We next compare the number of transactions missing from compact blocks received by the churning nodes M_i and X_i. **Figure** 3·17 shows the results obtained from the

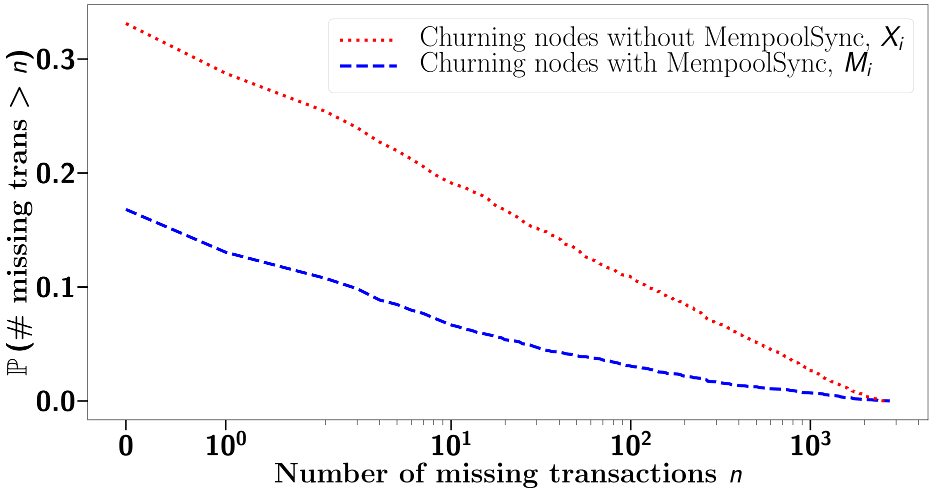


Figure 3·17: CCDF of number of missing transactions across all blocks for all nodes.

measurement. We find that churning nodes M_i that accept `MempoolSync` messages miss, on average, 21.30 transactions from blocks they received, with a standard deviation of 155.00 transactions. Churning nodes X_i that do not accept `MempoolSync` messages, on the other hand, miss, on average, 78.07 transactions from blocks they received, with a standard deviation of 288.04 transactions. While roughly 11% of blocks received by churning nodes X_i miss more than 100 transactions, just below only 3% of blocks received by churning nodes M_i miss more than 100 transactions. Similarly, a smaller fraction of blocks received by churning nodes M_i miss more than 1,000 transactions as compared to blocks received by churning nodes X_i. Thus, to a large degree, `MempoolSync` successfully synchronizes the mempools of churning nodes. This synchronization results in far fewer missing transactions in compact blocks.

3.3.5 Statistics on propagation delay

With a smaller number of transactions missing from the compact blocks, one can expect that churning nodes implementing `MempoolSync` will have a smaller block propagation delay than churning nodes not implementing `MempoolSync`. **Figure** 3·18

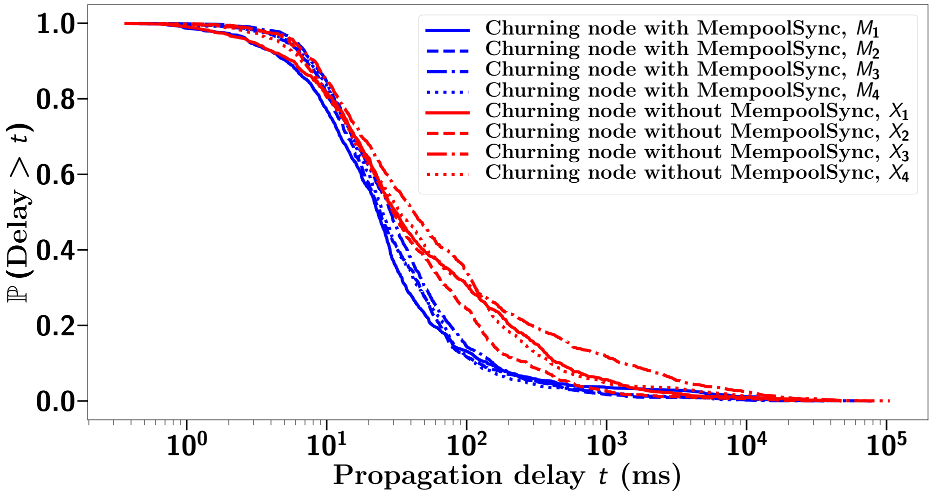


Figure 3·18: CCDF of propagation delay across all blocks for all nodes.

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

On an aggregate level, roughly 80% of blocks received by churning nodes X_i have a propagation delay larger than blocks received by churning nodes M_i. Indeed, on an aggregate level, roughly 30% of blocks received by churning nodes X_i experience a propagation delay larger than 100 ms with a maximum block propagation delay of 105.54 s. Comparatively, only about 12% of blocks received by churning nodes M_i experience a propagation delay larger than 100 ms with a maximum propagation delay of 78.83 s.

3.4 Discussions and limitations

Characterization of churn. Our characterization of churn in the Bitcoin network relies on the data obtained from Bitnodes which we assume to be accurate. To the best of our knowledge, Bitnodes does not discover nodes behind NAT or firewalls, and, therefore, the characterization is limited to behavior of nodes reachable by Bitnodes. Furthermore, it is not known what the intentions of these reachable nodes are. However, our data indicates that a large majority ($> 97.5\%$) announces access to the entire blockchain whereas more than 60% of the remaining nodes run in pruned-mode. Therefore, it is evident that these nodes take part in disseminating blocks through the network which can be affected when these nodes churn. Note that the Bitcoin Core is only *one* implementation of the Bitcoin protocol. It is worth noting that there are other implementations of the protocol, such as btcd [205], Bitcoin Knots [206, 207], libbitcoin [208], bitcoinj [209], etc., that do not fork the Bitcoin blockchain [210] but "speak" Bitcoin and are theoretically indistinguishable from one another.

We stress that archival nodes are necessary to allow new nodes to download the blockchain when they rejoin the network. Without such nodes, an adversary could force a new node to download a false blockchain. In addition, without non-miner nodes present in the network, it would become centralized in the sense that miners would have authority over consensus of new blocks.

It may be interesting to study the effects of churn in miners (specifically, mining pools) who may implement their own internal networks for faster block dissemination. Nodes operating in these mining pools, therefore, may not be reachable by Bitnodes and are, consequently, excluded from our characterization of churn. Churn could also be modeled as a function of the number of connections that a node has. A node with higher number of connections may affect more peers when it churns. However, we note that it is not easy to measure the number of connections of a node in the Bitcoin network without knowledge of the full network topology, which is kept intentionally hidden. Moreover, in our experiments, we do not artificially modify the number of connections of nodes from the default to avoid undesirable bias without prior knowledge of number of connections of other nodes in the network. These may be interesting follow ups to work presented in this chapter which we leave for future work.

Sampled session lengths. The session lengths sampled for our experiments are capped at a maximum of 1 day. However, to make sure that our results are statistically accurate, we sampled session lengths until we obtained a set of session lengths that had a mean within 1% of the mean of the original data set. We assume that nodes in the Bitcoin network exhibit a homogeneous churn behavior and follow the distributions obtained in **Section 3.1.3**. Note that the session lengths are independently sampled for all nodes in our experiment, and each node's sampled session lengths are also independent from one another as illustrated in **Figure 3·11** and **Figure 3·16**.

MempoolSync. We have implemented `MempoolSync` as a *proof-of-concept* to highlight the benefits of synchronizing mempools of churning nodes with highly-connected nodes in the Bitcoin network.

In our evaluation of `MempoolSync`, only one receiver was connected to a sender. We notice that with our current implementation, Bitcoin can easily handle the load of `MempoolSync` on a peer-to-peer basis. However, it is evident that in case of many churning nodes, one would need to implement a load balancing mechanism to avoid overload and network overhead at a single sender. While a majority of nodes in the network churn, we find that a large fraction of nodes does not churn as often as other nodes as shown in **Figure 3·2**. Nonetheless, there still remains a question of how churning nodes can identify highly-connected nodes in the network in a trustless and decentralized manner.

We specifically did not connect the churning nodes not configured with `MempoolSync` to control nodes as opposed to connecting churning nodes configured with `MempoolSync` to control nodes in our experiments. This is because we wanted to obtain data for regular churning nodes without interfering with how they discover and connect to peers. Connecting churning nodes not configured with `MempoolSync` to control nodes also creates an edge between the former and churning nodes configured with `MempoolSync` connected to the same control node. This may introduce undesirable bias in our data. It is also evident that since `MempoolSync` is not a two-way synchronization protocol, it may cause unnecessary network overhead if a receiver does not churn.

Finally, note that Bitcoin is not a stationary but dynamic system and overtime statistics *will* change. Hence, it is unclear whether running experiments over longer periods would provide more statistically meaningful data. Therefore, we believe our choice of running experiments over a period of two weeks is adequate. It should be noted that results obtained from our experiments are quite consistent across different categories of nodes, *i.e.,* control nodes, churning nodes not configured with `MempoolSync`, and churning nodes configured with `MempoolSync` as shown in **Sections 3.2** and **3.3.2**.

3.5 Summary

The main results from the work presented in this chapter are summarized below.

- We identified the previously unreported effects of churn on the Bitcoin network and performed a characterization of churn by fitting statistical distributions to the up and down session lengths.

- We emulated churn on full nodes connected to the *live* Bitcoin network to empirically demonstrate the effects of churn on the performance of the compact block relay protocol. Our experiments showed that churning nodes (i) fail to reconstruct a larger fraction of compact blocks; (ii) miss a large number of transactions from these blocks; and (iii) incur a larger delay in propagation of blocks than control nodes.

- We proposed and implemented a novel proof-of-concept synchronization scheme, `MempoolSync`, in the Bitcoin Core to mitigate the deteriorating effects of churn on the performance of the Bitcoin protocol. We show that by periodically synchronizing the mempool of churning nodes with those of control nodes, the performance of the compact block relay protocol can be significantly improved.

Our methodology can be extended to other blockchain systems to study the effects of churn on their underlying block relay protocols. Nonetheless, it is evident from our work that there is significant benefit in implementing efficient synchronization of the mempools of blockchain nodes, thus keeping them up-to-date with transactions that they might have missed while being disconnected.

Chapter 4

Empirical comparison of block relay protocols for blockchains

Our findings in **Chapter 3** show that the performance of the compact block relay protocol degrades when full nodes churn in the Bitcoin network. This subsequently raises the question whether other block relay protocols suffer from the same problem and how they compare one to another when full nodes churn in the corresponding blockchain networks. Churn is ubiquitous and pervasive in blockchain networks [68–70] and may occur due to a variety of reasons, such as the need to apply software patches or intermittency of power or network connectivity. Indeed, power outages are common in developing countries [79–86] and not unusual in developed countries as well [87–98].

To this effect, in this chapter, we analyze and quantify such real-world effects on three popular block relay protocols used in the Bitcoin ecosystem (described in greater detail in **Section 2.2.4**): (i) the legacy (*default*) block relay protocol, (ii) the *compact block* relay protocol [54], and (iii) the more recent *Graphene* [55, 123] block relay protocol. Our comparisons are carried out through the popular Bitcoin Unlimited (BU) client, which is a concrete implementation for Bitcoin Cash (a fork of the Bitcoin blockchain – see **Section 2.2.2**) that can support all three protocols after some code changes. Unlike existing simulation-based evaluations, our experiments include important real-world artifacts such as the fluctuations in node connectivity that are ubiquitous for these networks.

The rest of this chapter is organized as follows. In **Section 4.1**, we present empirical evaluations and compare one to another the performance of the three aforementioned block relay protocols. We take a deeper look in **Section 4.2** into the three block relay protocols and show insights relevant to the protocols. **Section 4.3** summarizes the main findings in the chapter.

4.1 Evaluation of block relay protocols

In this section, we experimentally evaluate the performance of the Graphene, compact, and default block relay protocols in three network regimes: (i) `always on`, which represents the ideal situation where full nodes stay continuously connected to the BU network; (ii) `statistical churn`, where nodes churn according to statistical characterization derived in **Section 3.1.3**; and (iii) `periodic churn`, whereby nodes follow a periodic churn pattern with fixed duration for the "on" and "off" periods. This allows us to better isolate churn factor that affect the performance of block relay protocols. Further, this can emulate the aforementioned disconnections due to power outages, with nodes staying off the network over extended durations.

The rest of this section is organized as follows. We first detail the mechanisms we employ to collect data from our experiments. Next, we describe our measurement setup. Then, in the rest of this section, we present statistics on the performance of the aforementioned block relay protocols in different network regimes.

4.1.1 Data collection mechanism

Our data collection mechanism builds upon the "log-to-file system" developed in **Section 3.2.1**. We significantly augment this logging system with new capabilities to record data relevant to Graphene blocks, including capturing various events relevant to this complex protocol, as described in **Section 2.2.4**. We also add new functionality so that data related to compact and default blocks can be recorded with finer granularity. The logging system allows one to (i) identify events when they occur; (ii) follow the changes in states as they take place; and (iii) record relevant data, *e.g.,* the hash of a block that is announced, the transactions in the block, etc., to files which one can later use to obtain results. The primary data point in our experiments is a block and we tie every data associated with the block to its unique hash. We do this for each Graphene, compact, and default block received by our measurement nodes. This allows us to isolate, identify, and acquire enough information to get necessary results. We use this logging system as our primary method of obtaining data in our experiments. Our expansion to the logging system is made public and is available on GitHub [124].

4.1.2 Experimental setup

The purpose of the experiments in this section is to gauge the performance of the Graphene, compact, and default block relay protocols. For this purpose, we connect 12 nodes to the live BU network. The nodes are Dell Inspiron 3670 desktops, each equipped with an 8th Generation Intel® Core i5−8400 processor (9 MB cache, up to 4.0 GHz), 1 TB HDD and 12 GB RAM. The nodes are each running the Linux Ubuntu 18.04.5 LTS (Bionic Beaver) distribution. The nodes run v1.9.0.1 of BU with an implementation of a bug fix [211] and an implementation of the logging system detailed in **Section 4.1.1**. We have made this version of BU public which is available on GitHub [124]. We emphasize that all of the nodes in our experimental setup are on both the `DST` side of **Listings 1** to **3** when they receive blocks *from* their peers and on the `SRC` side when they relay blocks *to* their peers. However, any reference to a “node” in this chapter is when it is on `DST` side and relevant information is recorded.

To study the performance of block relay protocols in the `always on` and `statistical churn` regimes, we run experiments and measure data over a period of two weeks starting from Tuesday, April 20, 2021 00:00:00 EST. Six nodes always stay connected to the BU network throughout the measurement period. Two of these nodes are configured to accept Graphene blocks only, two to accept compact blocks only, and the remaining two to accept neither, *i.e.,* accept default blocks only. Additionally, six nodes fluctuate on and off the BU network using session lengths sampled from distributions that represent churn in the Bitcoin network [69]. Specifically, the nodes stay *on* and *off* the network with session lengths sampled from the *log-logistic* (see **Equation (3.1)** and corresponding parameter values in **Section 3.1.3**) and the *Weibull* (see **Equation (3.2)** and corresponding parameter values in **Section 3.1.3**) distributions, respectively. Two of these nodes accept Graphene blocks only, two accept compact blocks only, and the remaining two only accept default blocks.

To study the performance of block relay protocols in the `periodic churn` regime, we introduce the following fluctuation periods: 20 minutes (m), 1 hour (h), 3 h, and 6 h. We chose the duration of these periods based on results of preliminary experiments: durations that are either shorter or longer do not yield results that are markedly different from ones presented in this section in the 20 minutes and 6 hours, respectively. For each of the fluctuation period duration, we further divide experiments into two cases. In the first experiment, which ran for one week starting from Thursday, March 25, 2021 22:00:00 EST, we set the off duty cycle to be 25% of the fluctuation period. That is, during each fluctuation period, the node stays off the network 25% of the time, and on the network for 75% of the time. In the second experiment, which ran for one week starting from Friday, April 02, 2021 02:01:19 EST, we similarly set the off duty cycle to be 75% of the total duration of the fluctuation periods. We split the 12 nodes in our testbed into four groups of three nodes. Each group is configured with one of the four aforementioned fluctuation periods (i.e., 20 m, 1 h, 3 h, and 6 h). Within each group, one node accepts Graphene blocks, one node accepts compact blocks, and the last node accepts default blocks only.

To get an in-depth view of additional transactions in `cmpctblock` messages, we performed another experiment which ran for two weeks starting from Friday, September 3, 2021 02:00:00 EST. In this experiment, three of the 12 nodes are always on, and the remaining nine statistically churn according to distributions presented in **Section 3.1.3**. Since this is a study on additional transactions in `cmpctblock` messages, all 12 nodes are configured to accept compact blocks only.

Note that nodes configured to accept Graphene or compact blocks only *must* also accept default blocks as a fail-over mechanism in case the aforementioned protocols fail drastically. Therefore, our testbed nodes could connect to peers that are not configured with the same relay protocol. In such cases, a testbed node and its peer will relay default blocks only. To make sure that we do not introduce any bias in our results, we do not force our nodes to drop connections with peers with whom they can only communicate via the default block relay. Results obtained from our experiment are detailed in the sections that follow. We have made our experimental logs publicly available for use by the wider research community [125].

4.1.3 Statistics on the propagation delay of blocks

In this section, we present statistics on the one-hop block propagation delays in different network regimes. We measure propagation delay as the difference between the time the header of a block, *i.e.,* the `headers` message, is received by a measurement node and the time at which the block is fully reconstructed and processed.

Figure 4·1 shows the complementary cumulative distribution function (CCDFs) of block propagation delays for nodes in the `always on` and `statistical churn` regimes configured with the Graphene, compact, and default block relay protocols. In nodes in the `always on` regime, Graphene, compact, and default blocks have mean propagation delays of 190.67 ms, 268.34 ms, and 974.11 ms, respectively, with stan-

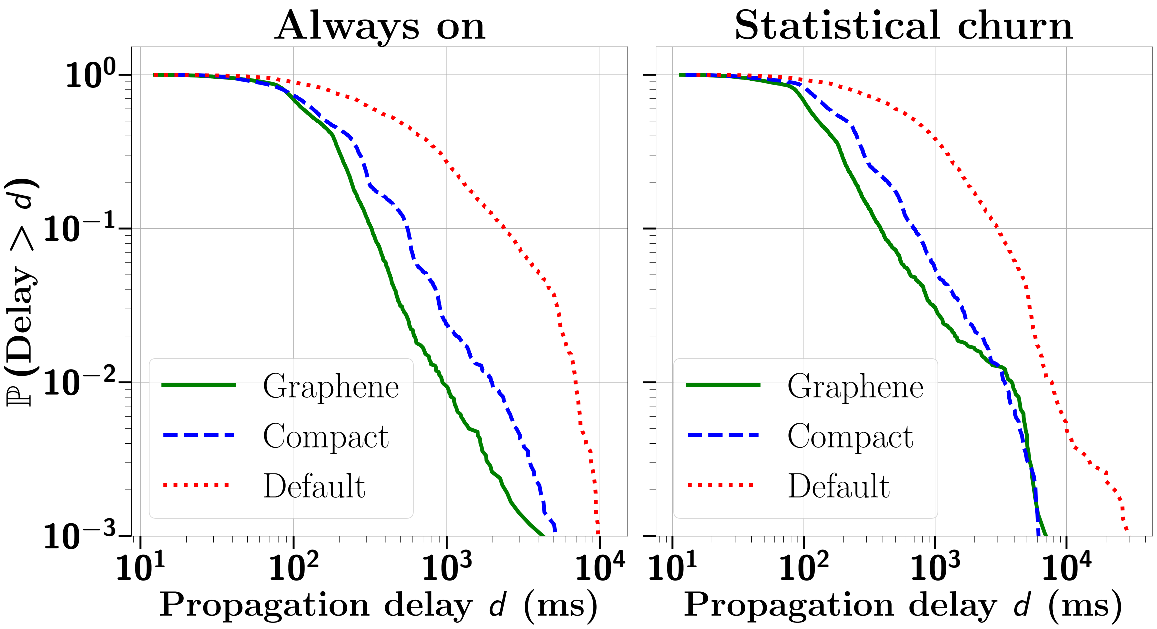


Figure 4·1: Complementary cumulative distribution functions (CCDFs) of block propagation delays in Graphene, compact, and default block relay protocols in the `always on` and `statistical churn` regimes. Graphene block relay protocol performs best in roughly 99% of blocks whereas default block relay protocol always performs worst.

dard deviations of 280.32 ms, 393.27 ms, and 1392.44 ms, respectively. On the other hand, in nodes in the `statistical churn` regime, Graphene, compact, and default blocks have mean propagation delays of 259.13 ms, 364.19 ms, and 1287.22 ms, respectively, with standard deviations of 649.99 ms, 718.58 ms, and 2357.11 ms, respectively.

These statistics show that (i) Graphene blocks have smaller average propagation delays out of the three protocols whereas default blocks have the largest average propagation delays across both regimes, and (ii) blocks across the three different protocols in the `statistical churn` regime always have, on average, larger block propagation delays as compared to blocks across the respective protocols in the `always on` regime which is explained by nodes not receiving several transactions from their peers while they are off the network. Upon receiving a block which may contain many of these missing transactions, the nodes must perform round-trip communication to recover

Fluctuating period	**Graphene**		**Compact**		**Default**	
	100 ms	**1,000 ms**	**100 ms**	**1,000 ms**	**100 ms**	**1,000 ms**
20 m	83.65%	10.57%	86.25%	16.57%	98.16%	69.19%
1 hr	76.56%	15.65%	84.10%	15.42%	95.88%	51.67%
3 hr	75.26%	8.85%	79.22%	8.62%	93.55%	37.67%
6 hr	75.60%	3.30%	79.03%	3.41%	93.60%	36.65%

(a)

Fluctuating period	**Graphene**		**Compact**		**Default**	
	100 ms	**1,000 ms**	**100 ms**	**1,000 ms**	**100 ms**	**1,000 ms**
20 m	95.66%	42.02%	96.72%	49.13%	99.19%	86.54%
1 hr	75.35%	28.46%	78.96%	21.99%	97.24%	46.81%
3 hr	78.86%	16.61%	77.99%	14.92%	96.34%	48.32%
6 hr	77.52%	9.22%	76.02%	9.03%	92.04%	38.75%

(b)

Table 4.1: Fraction of blocks that have propagation delay larger than 100 ms, and 1,000 ms in Graphene, compact and default block relay protocols over varying fluctuating periods with **(a)** 25%, and **(b)** 75% off duty cycles.

the transactions (in the case of Graphene and/or compact block relay protocols) and/or to recover from block decode failure (in the case of Graphene block relay protocol) which also results from missing transactions. The extra communication adds to the delay in reconstructing blocks and, consequently, the delay in propagation of these blocks.

Table 4.1 shows the fraction of blocks that have a propagation delay exceeding 100 ms and 1,000 ms across the block relay protocols in nodes in the `periodic churn` regime for different fluctuating periods and off duty cycles. A key takeaway from the table is that the default block relay protocol always performs worse than both the Graphene and compact block relay protocols. This is because the default block contains full transactions as compared to Graphene and compact blocks that contain short hashes of a majority of transactions. Therefore, default blocks take longer to propagate.

A trend that can be observed across all fluctuating periods is that both Graphene and compact block relay protocols perform worse when they are off the network for 75% of the fluctuating periods. Similar to the case with `always on` and `statistical churn` regimes, this can be attributed to extra round-trip communication for recovering missing transactions and performing failure recovery.

Next, it is apparent that as the lengths of the fluctuating periods increase, the performance of both the Graphene and compact block relay protocols improves. That is, a smaller fraction of blocks has large propagation delays. We theorize that this is because as the nodes stay off the network for longer, a large number of the transactions that they miss receiving from their peers is already included in blocks that they also miss receiving while they are off the network. Therefore, once they rejoin the network, they miss fewer transactions and suffer from fewer failures, thereby requiring fewer additional round-trip communications. Therefore, they experience shorter block propagation delays.

On the other hand, we note that the Graphene block relay protocol generally outperforms the compact block relay protocol except for a few cases, especially when the fluctuating period is 1 hr and the off duty cycle is 75% of the fluctuating period. This is likely because the node misses several transactions from its peers that are going to be included in the next few blocks that it will receive. Therefore, the node requires additional round-trip communication to recover these transactions adding to the propagation delay of blocks.

4.1.4 Statistics on the communication size per block

Next, we look at statistics on the communication size per block received by different types of nodes. When calculating the total communication size per block, we take into account the serialized size of the initial block received (*i.e.,* `grblk`, `cmpctblock`,

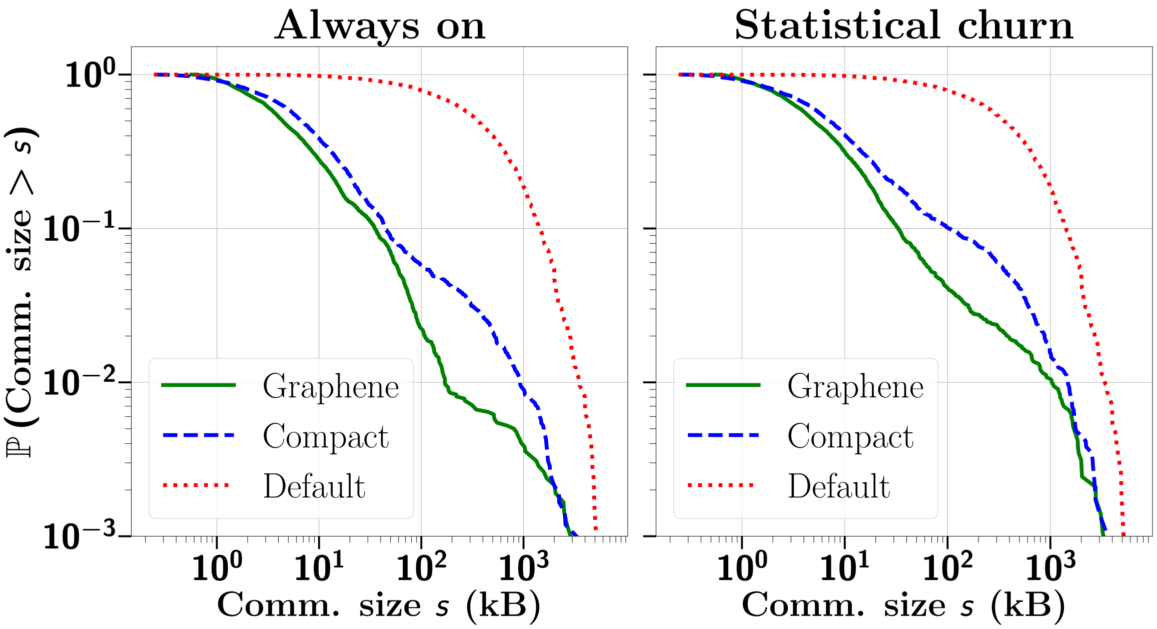


Figure 4·2: CCDFs of block communication sizes in Graphene, compact, and default block relay protocols in the *`always on`* and *`statistical churn`* regimes. Graphene block relay protocol performs best in both regimes whereas default block relay protocol performs worst.

and `block` for Graphene, compact and default blocks, respectively), and the serialized sizes of all follow up round-trip messages sent and received to and from peers. These messages could, *e.g.,* be sent to recover missing transactions from peers or to perform failure recovery. For detail on all possible message exchanges between nodes in BU, please refer to **Section 2.2.4**.

Figure 4·2 shows the CCDFs of block communication sizes for the nodes in the `always on` and `statistical churn` regimes configured with the Graphene, compact and default block relay protocols. In nodes in the `always on` regime, Graphene, compact, and default blocks have mean communication sizes of 24.53 kB, 46.17 kB, and 599.93 kB, respectively, with standard deviations of 183.82 kB, 218.79 kB, and 716.71 kB, respectively. On the other hand, in nodes in the `statistical churn` regime, Graphene, compact, and default blocks have mean communication sizes of 40.58 kB, 73.37 kB, and 592.64 kB, respectively, with standard deviations of 234.06 kB,

Block relay protocol	**Comm. size**	**Fluctuating period**			
		20 m	**1 hr**	**3hrs**	**6 hrs**
Graphene	**10 kB**	57.20%	69.41%	50.82%	51.54%
	100 kB	47.72%	54.45%	23.04%	13.06%
	1,000 kB	23.90%	23.54%	9.04%	4.70%
Compact	**10 kB**	76.65%	64.40%	57.92%	51.50%
	100 kB	52.29%	52.05%	26.76%	13.53%
	1,000 kB	18.27%	13.93%	9.18%	5.10%
Default	**10 kB**	98.56%	98.50%	98.46%	99.23%
	100 kB	82.71%	81.05%	81.37%	81.03%
	1,000 kB	30.51%	25.98%	26.62%	28.63%

(a)

Block relay protocol	**Comm. size**	**Fluctuating period**			
		20 m	**1 hr**	**3hrs**	**6 hrs**
Graphene	**10 kB**	26.93%	29.91%	41.41%	31.74%
	100 kB	11.20%	18.83%	12.46%	7.77%
	1,000 kB	3.39%	11.07%	4.75%	2.13%
Compact	**10 kB**	63.20%	60.83%	57.26%	53.20%
	100 kB	25.10%	24.46%	19.45%	9.55%
	1,000 kB	5.87%	4.77%	5.53%	1.81%
Default	**10 kB**	98.15%	98.15%	98.09%	98.05%
	100 kB	83.03%	82.46%	82.14%	81.72%
	1,000 kB	31.54%	28.97%	28.57%	27.79%

(b)

Table 4.2: Fraction of blocks that have communication sizes larger than 10 kB, 100 kB, and 1,000 kB in Graphene, compact and default block relay protocols over varying fluctuating periods with **(a)** 25%, and **(b)** 75% off duty cycles.

279.07 kB, and 701.68 kB, respectively.

Similar to **Section** 4.1.3, the rise in communication sizes of blocks received by nodes in the `statistical churn` regime compared to blocks received by nodes in the `always on` regime can be attributed to extra round-trip communication needed to recover missing transactions and perform failure recovery.

Table 4.2 shows the fractions of blocks that have a total communication size larger than 10, 100, and 1,000 kBs across the block relay protocols in nodes in the `periodic churn` regime for different fluctuating periods and off duty cycles. Similar to the statistics presented in **Section 4.1.3**, a key takeaway from the table is that the default block relay protocol always performs worse than Graphene and compact block relay protocols. That is, default blocks are always larger in size than Graphene and compact blocks and any additional round-trip communication *combined*. This is because default blocks contain full transactions each of which can be several hundred bytes in size [212, 213]. By comparison, Graphene and compact blocks contain only short hashes representing transactions which considerably reduce the overall size of the blocks.

We observe more trends similar to those highlighted in the **Section 4.1.3**: across all fluctuating periods, both Graphene and compact block relay protocols perform worse when their off duty cycle is 75% of the fluctuating period; as the length of the fluctuating periods increase, the performance of the Graphene and compact block relay protocols improves. Finally, the Graphene block relay protocol almost always outperforms the compact block relay protocol except in a few cases where our proposed theory from the previous section applies.

4.1.5 Correlation between propagation delay and communication per block

As observed from the previous two sections, the communication and delay performance of the various protocols follow similar trends. We next rigorously quantify the correlation between these metrics by calculating the Spearman Rank Correlation (SRC) coefficient ρ [214].

Provided two data sets D_1 and D_2 of equal size n, the SRC coefficient is given by

$$\rho = 1 - \frac{6\sum_{i=1}^{n}\left(D_{1_i} - D_{2_i}\right)^2}{n\left(n^2-1\right)},$$

where $-1 \leq \rho \leq +1$, and D_{1_i} and D_{2_i} are the *ranks* of the i^{th} data point in sets D_1

Block relay protocol	**Graphene**	**Compact**	**Default**
Correlation coefficient ρ	0.51	0.61	0.68

Table 4.3: Coefficients for Spearman Rank Correlation between the block propagation delays and block communication sizes in Graphene, compact, and default block relay protocols in nodes in the `statistical churn` regime.

Fluctuating period	ρ					
	Graphene		**Compact**		**Default**	
	25%	**75%**	**25%**	**75%**	**25%**	**75%**
20 m	0.69	0.76	0.68	0.78	0.67	0.50
1 hr	0.65	0.85	0.69	0.84	0.50	0.42
3 hr	0.60	0.70	0.72	0.79	0.61	0.32
6 hr	0.56	0.61	0.72	0.71	0.61	0.74

Table 4.4: Coefficients for Spearman Rank Correlation between the block propagation delays and block communication sizes in Graphene, compact, and default block relay protocols in nodes in the `periodic churn` regime. In general, the propagation delays and communication size are moderately to highly correlated.

and D_2, respectively [215]. Values of $\rho = +1$ and $\rho = -1$ imply an exact monotonic relation between data sets D_1 and D_2 where the former implies that D_1 increases as D_2 increases and the latter implies the opposite [216]. The ranks in SRC are determined as follows: the data sets are sorted in ascending order and the values are replaced by their corresponding ranks [217].

Our results are summarized in **Tables 4.3** and **4.4**. We find that there generally exists moderate, *i.e.*, $\rho \in [0.4, 0.6)$, to strong relationship, *i.e.*, $\rho \in [0.6, 0.8)$ [218], between block propagation delays and block communication sizes in all three block propagation relay protocols. That is, as churning nodes exchange additional messages to recover missing transactions in blocks, their propagation delay can be expected to increase regardless of the geographical locations of neighboring peers in the Bitcoin network.

4.2 Insights into block relay protocols

In this section, we present extensive analyses and insights into the three block relay protocols. We take a deep dive into the causes of the deteriorated performance of the Graphene block relay protocol in **Section** 4.2.1. We study the events that occur corresponding to the Graphene blocks received by the measurement nodes in a temporal analysis presented in **Section** 4.2.2. In **Section** 4.2.3, we compare the sizes of the first messages relayed in each protocol. We present an analysis of the usefulness of the additional full transactions in the `cmpctblock` messages in **Section** 4.2.4.

4.2.1 Graphene in depth

Our findings in **Sections** 4.1.3 and 4.1.4 indicate that the performance of the Graphene block relay protocol degrades in some cases. In this section, we take a deeper look into the reasons that cause this degraded performance.

Recall from **Section** 2.2.4 that there are several scenarios in which the Graphene protocol requires extra round-trip communication which includes recovering missing transactions from peers and performing failure recovery when block decode fails. The Graphene block relay protocol is complex: block decode could be successful but there might be transactions missing from the mempool. On the other hand, block decode could fail and there may or may not still be transactions missing from the mempool even after failure recovery.

Block decode failure occurs when the condition in line 8 of **Listing** 3 returns `false`. That is, the subtraction operation $I - I'$ fails. When this happens, failure recovery is performed which is depicted by lines 17 onward, *i.e.,* scenarios ③, ④, and ⑤, in **Listing** 3.

Figure 4·3 shows the proportion of Graphene blocks that suffer from decode failure when the off duty cycle is 25% and 75%, respectively, and for different fluctua-

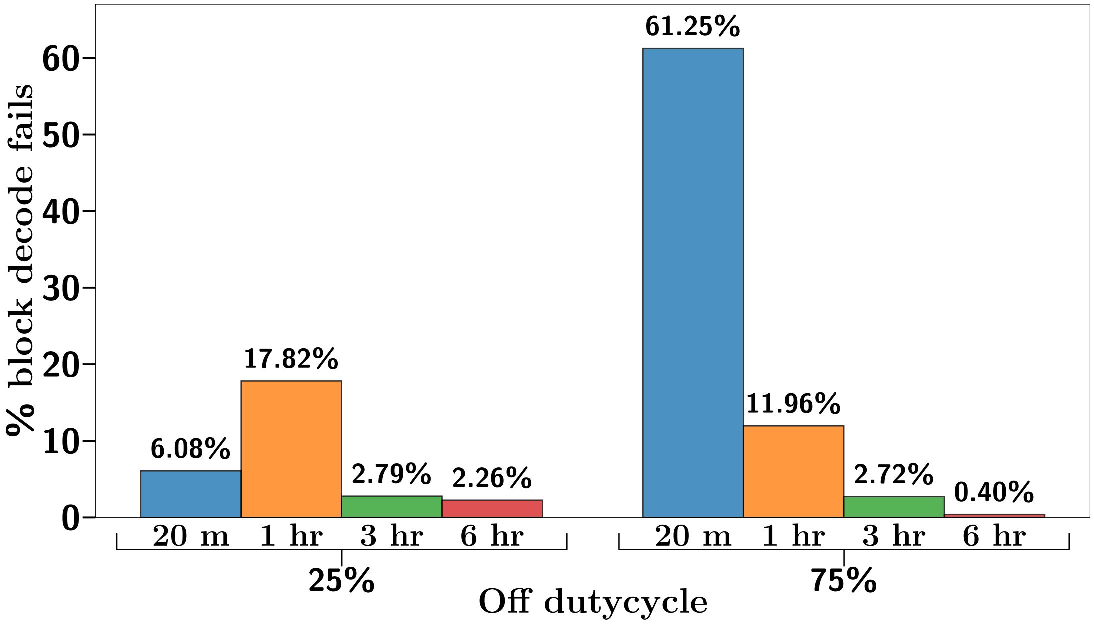


Figure 4·3: Proportion of block decode failures, *i.e.*, scenarios ③, ④, and ⑤ in **Listing** 3, over different fluctuation periods with 25% and 75% off duty cycles. Block decode failure rates are higher when nodes churn more often and stay off the network longer thereby not being able to recover. This is prominent in fluctuating periods of 20 m and 1 hr.

tion periods. The figure shows that block decode failure rates are higher when nodes churn more often *and* stay off the network longer. They are, hence, unable to recover from staying off the network. As the nodes churn less frequently as well as stay on the network longer, the block decode failure rates significantly drop.

Next, we investigate the case when block decode is successful but transactions are missing from the mempool of the node when a new block is received. This is represented as Scenario ② in **Listing** 3 on lines 12-16.

Figure 4·4 shows the mean number of transactions missing from mempool when a block is received and successfully decoded with 25% and 75% off duty cycle. The combination of **Figures** 4·3 and 4·4 provide a thought-provoking insight: when a node churns frequently, it misses receiving enough transactions that will result

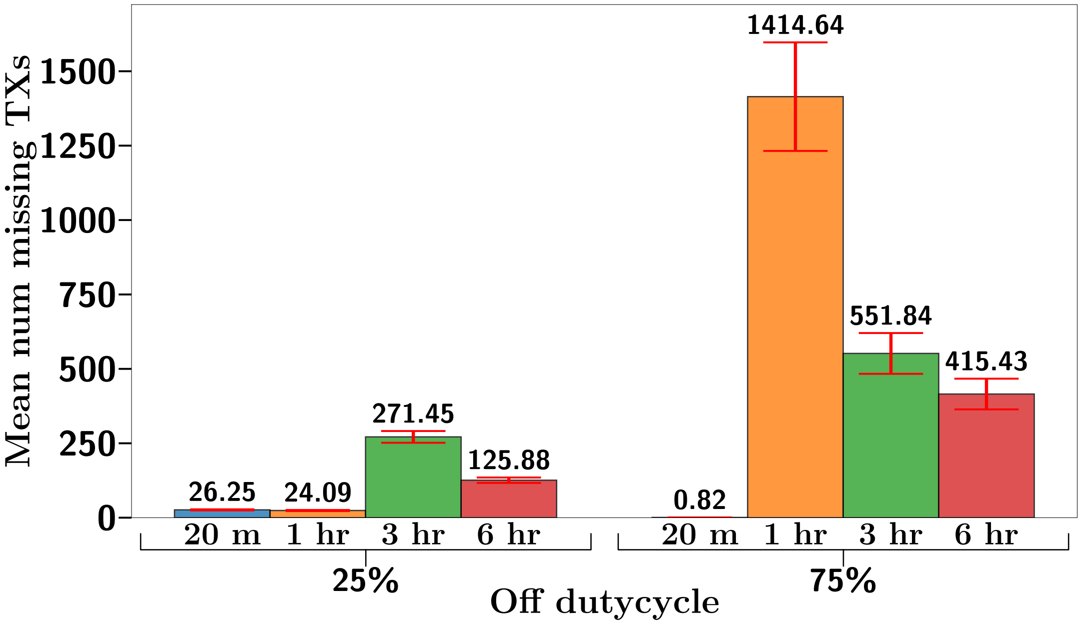


Figure 4·4: Average number of missing transactions (with 95% confidence intervals) from blocks that are decoded successfully, *i.e.*, scenario ② in **Listing** 3 over different fluctuation periods with 25% and 75% off duty cycles.

in a higher fraction of block decode failures and the missing transactions will be recovered via failure recovery. On the other hand, when a node churns less frequently, it still misses transactions which are not enough to cause block decode failure. These transactions are then recovered by sending transaction recovery requests to peers. Additionally, as the nodes stay on the network for longer, they miss fewer transactions. In either cases, *both* failure recovery and recovering missing transactions require an extra round-trip communication.

Finally, we take a look at the case when block decode is unsuccessful *and* there are still missing transactions even after failure recovery is performed. This is represented as scenario ④ in **Listing** 3 on lines 29-33. Interestingly, **Figure** 4·5 reveals that when nodes fluctuate frequently *and* stay off the network longer, they may *still* miss transactions after failure recovery. This requires an additional extra round-trip of communication *on top of* that needed to perform failure recovery.

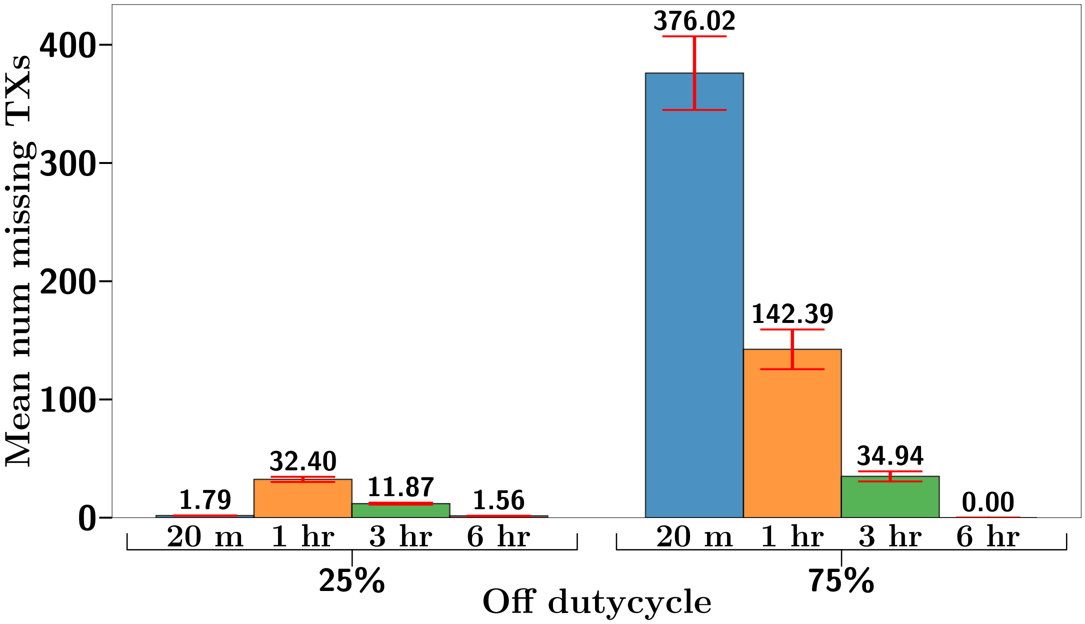


Figure 4·5: Average number of missing transactions from blocks that are decoded successfully, *i.e.*, scenario ④ in **Listing** 3 over different fluctuation periods with 25% and 75% off duty cycles.

The insights presented in this section explain the cause behind degraded performance of the Graphene block relay protocol in the case when nodes churn frequently and stay off the network longer. In some cases, up to two extra round-trips of communication are required resulting in higher delays in propagation and larger communication sizes.

4.2.2 Temporal analysis of the Graphene block relay protocol

In this section, we study the prevalence of scenarios discussed in **Section** 2.2.4 in Graphene blocks received by churning nodes. That is, what happens in the Graphene block relay protocol when a churning node rejoins the network. For this purpose, we create collections of blocks received in each interval for which the churning nodes are connected to the Bitcoin Unlimited network. We then identify the scenarios that each block goes through.

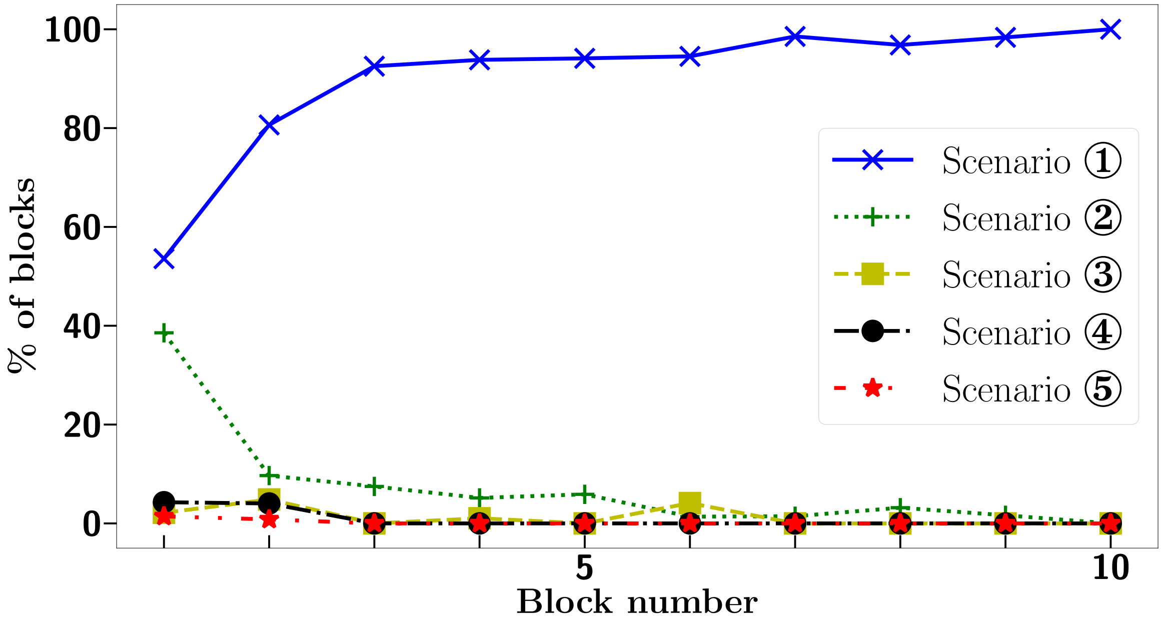


Figure 4·6: Percentage of blocks received in the `statistical churn` regime that face the five scenarios after the churning nodes rejoin the network. The longer a node stays on the network, the more scenario ① (i.e., no extra round-trip) prevails whereas the scenarios ③, ④, and ⑤ do not occur very often.

We first consider the `statistical churn` regime. **Figure** 4·6 shows findings for the first 10 Graphene blocks received after rejoining the network. Roughly 54% of the Graphene blocks received by a node immediately after it rejoins the network are successfully decoded and have no missing transactions. A significant portion, *i.e.,* roughly 39%, of the Graphene blocks are successfully decoded but have missing transactions. This shows that nodes need to perform round-trip communication with their peers to recover missing transactions immediately after they rejoin the network. While there are some blocks that suffer from scenarios involving block decode failures, their proportion is relatively small. As the node stays connected to the network and receives further blocks, the chances of transactions missing from the block significantly decrease. This trend continues on albeit some random off shoots with small probability (depicted by small peaks in the figure) in blocks with missing

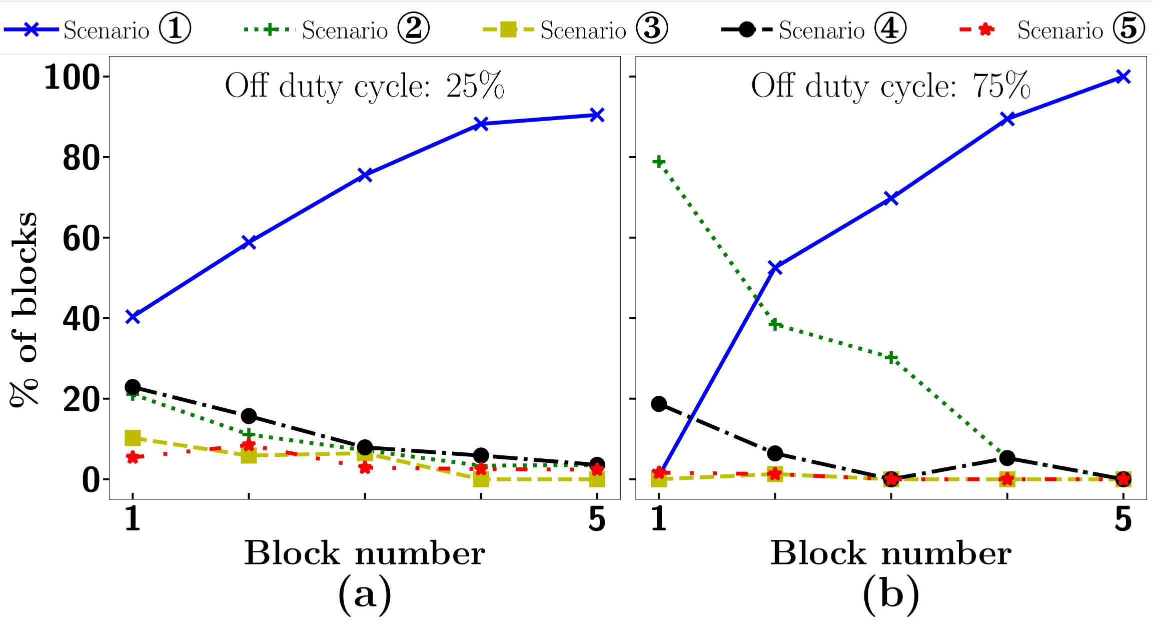


Figure 4·7: Percentage of blocks received in the `periodic churn` regime with a fluctuation period of 1 hr and off duty cycle of **(a)** 25% and **(b)** 75% that face the five scenarios after a node rejoins the network. In either case, scenario ① does not represent the majority of cases for the first block and scenarios ③ and ⑤ occur infrequently.

transactions.

We observe similar behavior in the `periodic churn` regime. For example, **Figure** 4·7**(a)** and **Figure** 4·7**(b)** show statistics for Graphene blocks received by nodes with a fluctuating period of 1 hr and off duty cycles of 25% and 75%, respectively. In both cases, when nodes rejoin the network, they see a large portion of blocks with missing transactions regardless of whether the IBLT decoding process is successful or not. This behavior is worse in nodes that stay off the network longer. However, similar to the `statistical churn` regime, the performance of the Graphene block relay protocol improves over time, and the proportion of blocks with missing transactions decreases significantly.

These results show that the performance of Graphene degrades when a node rejoins the network. However, as the node stays on the network, the protocol recovers. We believe this behavior motivates the necessity for a one-time mempool synchronization between peers when a churning node rejoins the network.

4.2.3 Size of first message across block relay protocols

In this section, we investigate the size of the first block message for each of the relay protocols as a function of the number of transactions included in a block. Initial messages are important because they are transmitted regardless of the specific scenario that ends up happening with future messages. We are specifically interested in identifying trends as well as outliers. Recall from **Section 2.2.4** that the first messages are `grblk`, `cmpctblock`, and `block` for the Graphene, compact, and default block relay protocols, respectively.

Figure 4·8 shows on the x-axis the number of transactions in a block received by nodes and on the y-axis the size of the first block message in bytes, in the `statistical churn` regime. *Notice that both the x- and y-axes are plotted on a log scale.* The figure shows that for default blocks, the `block` message is almost always the largest. This is because in default blocks, the first message *is* the entire block and contains full transactions included in the block. Hence, as the number of transactions in the block increase, so does the size of the `block` message. The `block` message has, on average, a size of 5.93×10^5 bytes with a standard deviation of 7.02×10^5 bytes.

Next we compare the sizes of `grblk` and `cmpctblock` messages. We observe that when the number of transactions in a block is small (*i.e.,* up to 60 transactions), the `cmpctblock` message *usually* has a smaller size than the `grblk` message. Further, there is a visibly direct relationship between the number of transactions in a compact block and the size of the `cmpctblock` message in the shape of an almost straight line. This line also forms a lower bound for the size of the `cmpctblock` messages, because for every transaction in a block, the `cmpctblock` message contains a 6-byte hash for the transaction. Therefore, as the number of transactions in a block increase, so does the size of the `cmpctblock` message. Note, however, that there are instances in which the size of the `cmpctblock` message deviates from the straight line.

We conjecture that this is due to the additional transactions included in the `cmpctblock` message. To verify this conjecture, **Figure** 4·9 shows on the x-axis the number of transactions in a block, on the left y-axis the size of the `cmpctblock` messages in bytes and on the right y-axis the number of additional transactions in `cmpctblock` messages. We observe a direct correlation between the number of additional transactions in the `cmpctblock` message and its size. That is, when the `cmpctblock` message contains *only* the coinbase transaction, its size follows the arithmetic progression mentioned earlier. However, the size of the `cmpctblock` message increases with the number of additional transactions it includes.

While it may appear from **Figure** 4·8 that `cmpctblock` are smaller in size than the `grblk` messages, we emphasize that the former are only bounded from below by the straight line marked by crosses in **Figure** 4·8. Sizes of `grblk` messages, on the other hand, appear to be bounded from above by a curve, marked by green circles, that looks asymptotically linear. As the number of transactions in blocks increase, the sizes of the initial `grblk` message tend to significantly deviate from the curve. We find contrary to `cmpctblock` messages that the sizes of `grblk` messages do not depend on the number of additional transactions in the message. Upon examining software implementation of the Graphene protocol in Bitcoin Unlimited, we find that `grblk` messages always contain only one additional transaction: the coinbase transaction. Therefore, we conjecture that the deviation from the curve is best explained by the size of the mempool sent by the `SRC` node to the `DST` node (see the discussion in **Section** 2.2.4). This parameter determines the sizes of the Bloom filter and IBLT included in the `grblk` message which in turn determines the overall size of the message. We leave further examination of this conjecture to future work.

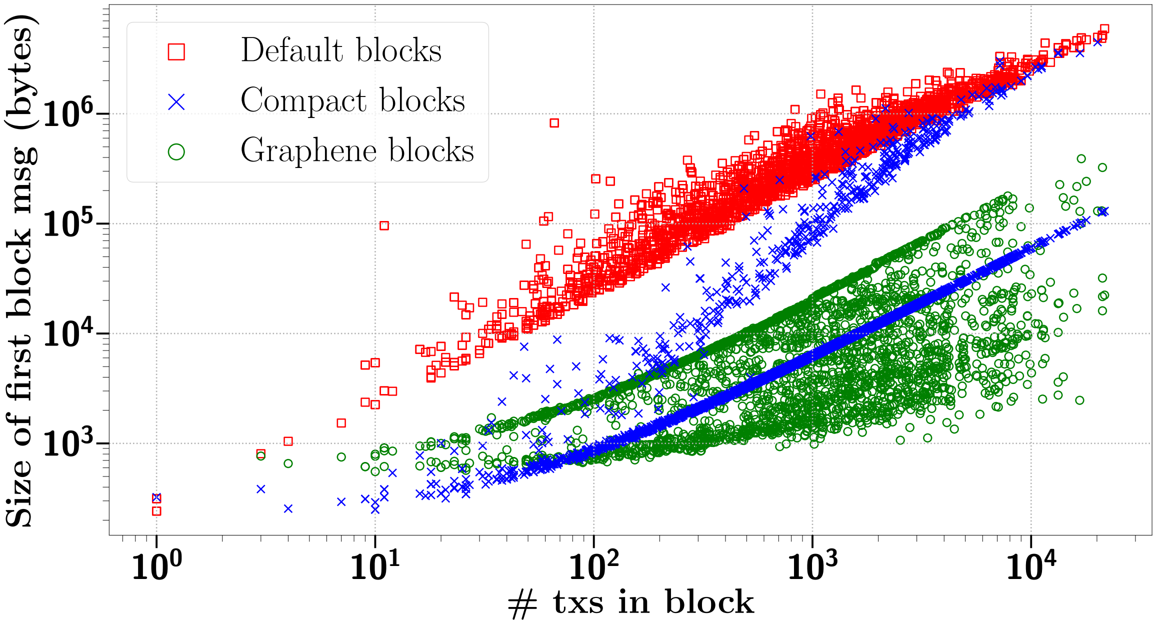


Figure 4·8: Sizes of the first block messages, *i.e.,* `block`, `cmpctblock`, and `grblk`, against the number of transactions in the respective compact blocks. The `block` messages almost always have the largest sizes.

Overall, the `grblk` message has an average size of 1.21×10^4 bytes with a standard deviation of 2.31×10^4 bytes, while the `cmpctblock` message has an average size of 6.24×10^4 bytes with a standard deviation of 2.60×10^5 bytes. Thus, the first message in Graphene is significantly smaller. Still, the `cmpctblock` message is much smaller than the `block` message of the default protocol. The latter has an average size of 5.93×10^5 bytes with a standard deviation of 7.02×10^5 bytes.

4.2.4 On the usefulness of additional transactions in the compact block relay protocol

Recall that the compact block protocol sends additional transactions as part of the `cmpctblock` messages. These are *full* transactions that the source node `SRC` predicts the receiving node `DST` may be missing from its mempool. In this section, we examine if these additional transactions are useful at all.

Denote by S_1 the set of additional transactions included in a block received by a

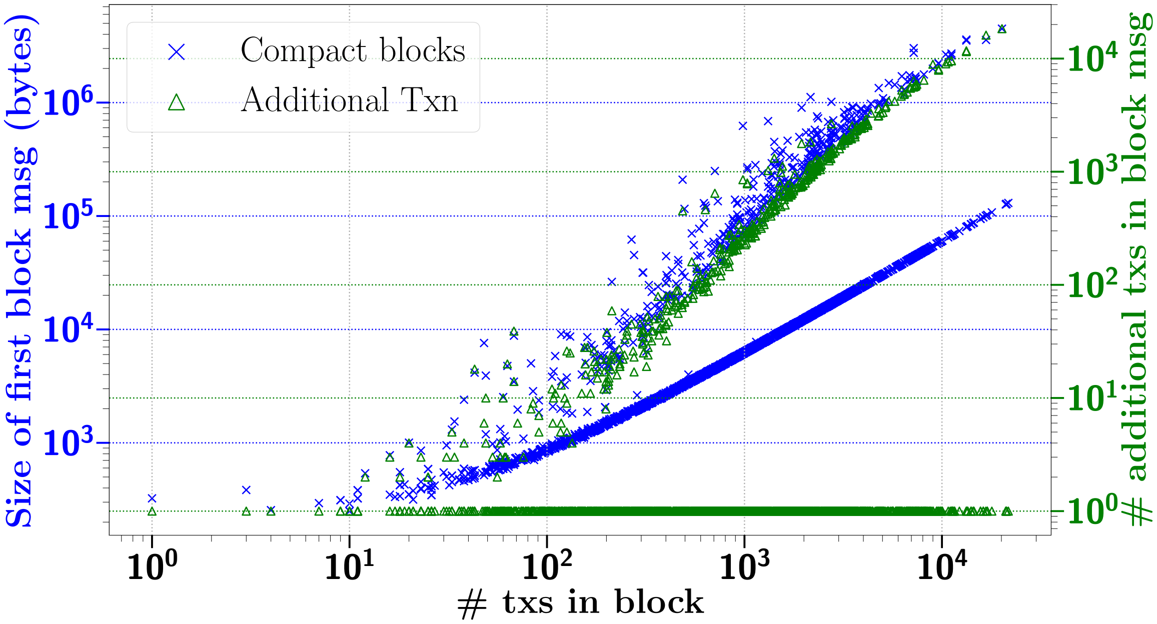


Figure 4·9: Sizes of the `cmpctblock` messages (left y-axis) and the number of additional transactions in blocks (right y-axis) against the number of transactions in the respective compact blocks. There exists a direct correlation between the number of additional transactions in and the sizes of the `cmpctblock` messages

node, and by S_2 the set of transactions contained in the mempool of the node when the block is received. The *number* of *useful* additional transactions, *i.e.,* transactions included in S_1 but not in S_2, is given by $\mid S_1 \setminus S_2 \mid$ where $\setminus$ denotes set subtraction.

The statistics on number of useful additional transactions in `cmpctblock` messages are shown in **Figure** 4·10. The x-axes show the number of additional transactions in a `cmpctblock` message and the y-axes show the number of useful additional transactions. The straight $x = y$ line represents 100% useful additional transactions in the `cmpctblock` message. The figure is divided into two halves where the upper half represents statistics in the `statistical churn` regime and the lower half in the `always on` regime. *Note that both x-axes and y-axes are plotted on a log scale.*

It can be observed from the figure that churning nodes have *relatively* more instances of useful additional transactions compared to always on nodes. This is because the former are likely to miss more transactions from their mempool than the latter since they were off the network. However, in both type of nodes, `cmpctblock` messages rarely have 100% useful additional transactions. In fact, in many cases, the *only* useful additional transaction in both churning and always connected nodes is the coinbase transaction, regardless of the number of additional transactions in the `cmpctblock` message. The data point clusters in the middle of the two halves of the figures show that in a few cases, some additional transactions other than the coinbase transactions are useful. However, even in those cases, the difference between the number of additional transactions and the number of useful additional transactions is usually high (up several orders of magnitude).

We next investigate whether useful additional transactions in `cmpctblock` messages save round-trip communication. That is, does the `DST` still misses transactions after receiving useful additional transactions included in a `cmpctblock` message? We next present statistics to answer these questions.

Denote by M the set of transactions in a node's mempool when it receives a block, by T the set of missing transactions in the block, and by A the set of additional transactions in the `cmpctblock` message for that block. Then $X = T \setminus M$ is the set of transactions in the block that are missing from the node's mempool, and $Y = A \cap (T \setminus M)$ is the set of additional transactions that help recover the missing transactions.

Our analysis shows that in both churning and always connected nodes, roughly 87% of `cmpctblock` messages contain only *one* additional transaction, *i.e.,* the coinbase transaction which, is always helpful. In the remaining `cmpctblock` messages, we exclude the coinbase transaction for further analysis. There are now three cases: a) the additional transactions are not helpful at all, *i.e.,* $|Y| = 0$; b) additional transactions are *partially* helpful, but not enough to recover all transactions in X, *i.e.,* $Y \neq X$ given $|Y| > 0$; or c) additional transactions are *completely* helpful and recover

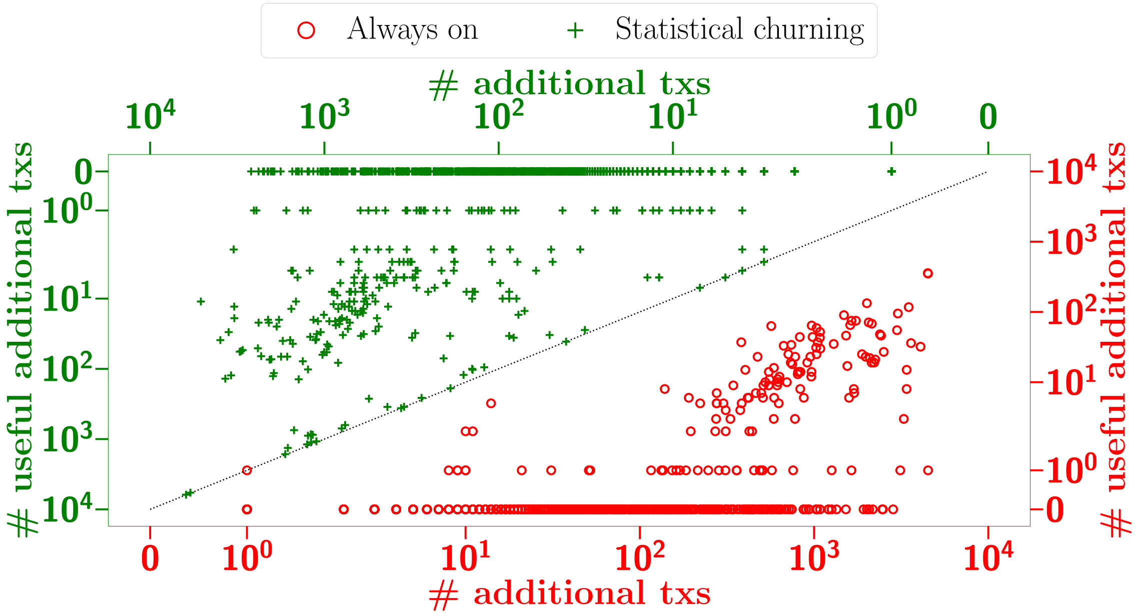


Figure 4·10: Number of useful additional transactions in `cmpctblock` messages against the total number of additional transactions in respective `cmpctblock` messages in (lower half) always on and (upper half) statistically churning nodes. The diagonal ($x = y$) represents the case when 100% of the additional transactions in `cmpctblock` messages are useful.

all transactions in X, *i.e.,* $Y \equiv X$. We find that there are no instances where the additional transactions are *completely* helpful to recover all transactions in X. On the other hand, additional transactions in roughly 84% of `cmpctblock` messages are *not* helpful *at all.* In the remaining roughly 16% of `cmpctblock` messages, additional transactions are *partially* helpful, *i.e.,* not enough to recover all transactions in X.

Our findings in this section show that while additional transactions (excluding the coinbase transaction) are sometimes helpful, in many instances they are either duplicates and thus end up wasting bandwidth, or not enough to completely recover transactions in blocks that may be missing in the node's mempool. Our calculations show that in the `always on` and `statistical churn` regimes, roughly 90% and 96%, respectively, of bandwidth consumed by additional transactions is unnecessary and wasted.

4.3 Summary

The main results from the work presented in this chapter are summarized below.

- We have empirically demonstrated and compared the performance of the Graphene, compact, and default block relay protocols in full nodes connected to the *live* Bitcoin Cash network via an instrumentation-capable version of the Bitcoin Unlimited client in three different network regimes.
- We show that in the `always on` and `statistical churn` regimes, the Graphene block relay protocol *always* performs better than the compact and default block relay protocols which respectively have roughly 40% and 500% higher propagation delays and over 80% and 150% larger communication sizes.
- We find that in the `periodic churn` regime, the Graphene block relay protocol *generally* performs better than the compact and default block relay protocols except for a few cases in which the compact block relay protocol performs better. We take a deep dive into why this happens and discover that frequently churning nodes may need to perform as many as *two* extra round-trips of communication to recover information necessary to successfully reconstruct Graphene blocks.
- We perform an in-depth analysis of the `cmpctblock` message in the compact block relay protocol and identify an inefficiency such that in many cases, the additional full transactions included in the message are always either not useful at all for or not enough for successful reconstruction of compact blocks resulting in an over 90% wasted bandwidth.

Based on the results obtained in this chapter, it seems preferable to configure nodes with the Graphene block relay protocol under typical network conditions. However, it may be more beneficial for frequently churning nodes to be configured with the compact block protocol. The methodology employed in this chapter can be further extended to other block relay protocols in different blockchains for a much wider comparison between the protocols.

Chapter 5

Orphan Transactions in the Bitcoin Network

In **Chapter 3**, we presented an extensive study on the impact of *churn* – an effect created by the independent arrival and departure of nodes in a peer-to-peer network [99]. We noticed during our measurement campaigns that some transactions ended up becoming *orphan*, that being transactions whose parental income sources are not known to full nodes upon receiving such transactions (see **Section 2.3** for details). 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 [219–225], security of transactions [226–231], studies on transaction confirmation times [232–235], 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 [40], and for denial of service attacks on the Bitcoin network [41, 42]. 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? Is there a connection between churning of a full node and the transactions it receives becoming orphan? There exists no work, to the best of our knowledge, that reasonably answers these questions.

In this chapter which is based on our work published in the proceedings of the *IEEE International Conference on Blockchain and Cryptocurrency 2020* [38] for which we also won the **Best Paper Award** [236], and the *IEEE Transactions on Network and Service Management* [39], 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.

The rest of this chapter is organized as follows: In **Section 5.1**, 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 5.2**. In **Section 5.3**, 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 5.4**. **Section 5.5** summarizes the main findings in the chapter.

5.1 Characterization of orphan transactions

We 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 the number of parents of all non-orphan transactions. Afterwards, 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.

5.1.1 Measurement setup

For **Sections** 5.1.2 to 5.1.5, 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 [237] on the Linux Ubuntu 18.04.2 LTS distribution, running on Dell Inspiron 3670 desktops, each equipped with an 8th 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 relevant extensions to the log-to-file system described in **Section** 3.2.1, for roughly 2 weeks over two rounds (November 18, 2019 11:00 AM EST to November 25, 2019 10:59 AM EST, and November 25, 2019 11:00 AM EST to December 02, 2019 10:59 AM EST).

In **Sections** 5.1.6 to 5.1.8, 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 EST for two weeks for **Sections** 5.1.6 and 5.1.7, and from November 11, 2020 1:30 PM EST for roughly one week for **Section** 5.1.8.

5.1.2 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

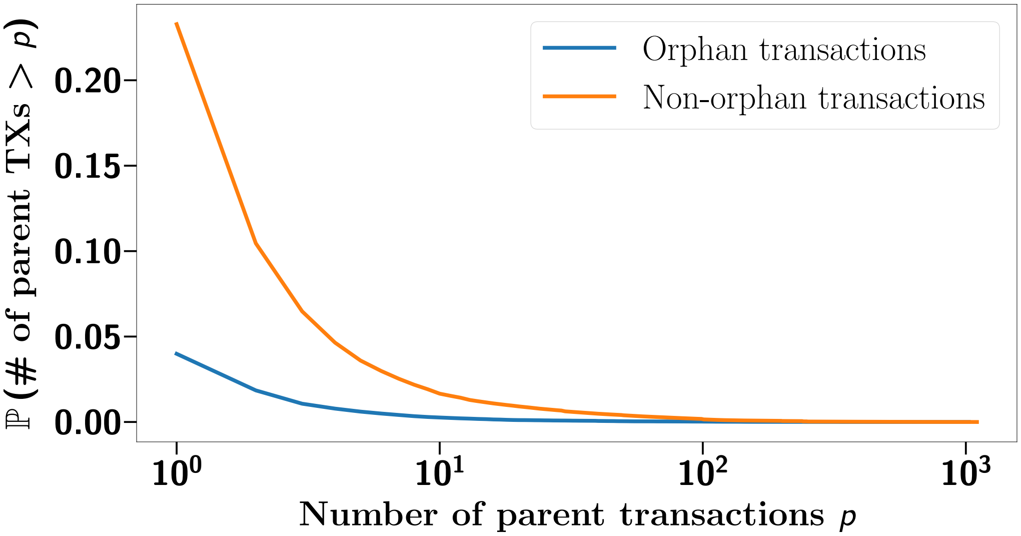


Figure 5·1: Empirical complementary cumulative distribution function (CCDF) of (i) the number of parents of orphan transactions and (ii) number of parents of non-orphan transactions. In general, orphan transactions have fewer parents.

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.

Figure 5·1 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 are 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

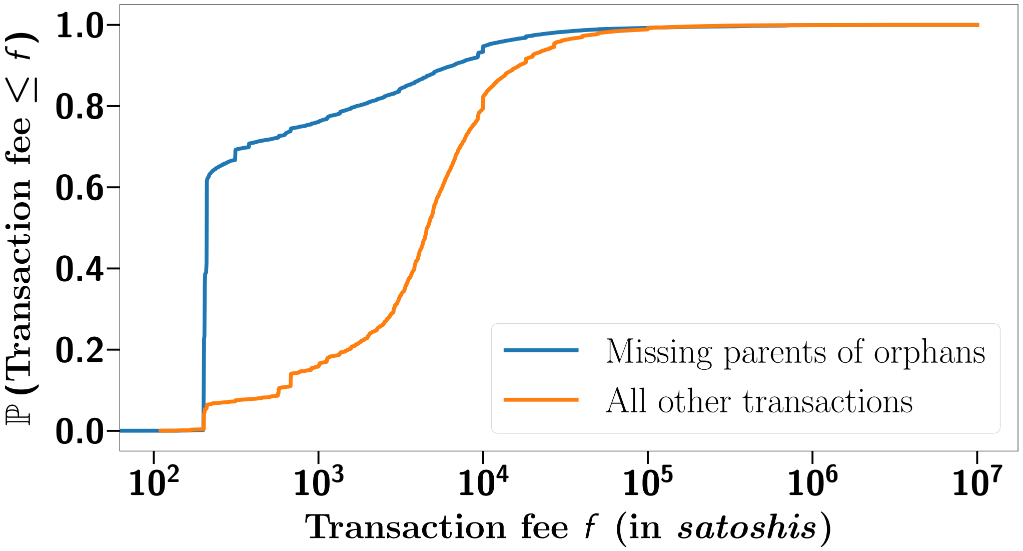


Figure 5·2: Cumulative distribution functions (CDFs) of transaction fee of missing parents of orphan transactions, and transaction fee of all other transactions.

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.

5.1.3 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.

Figure 5·2 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 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.

5.1.4 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?

Figure 5·3 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

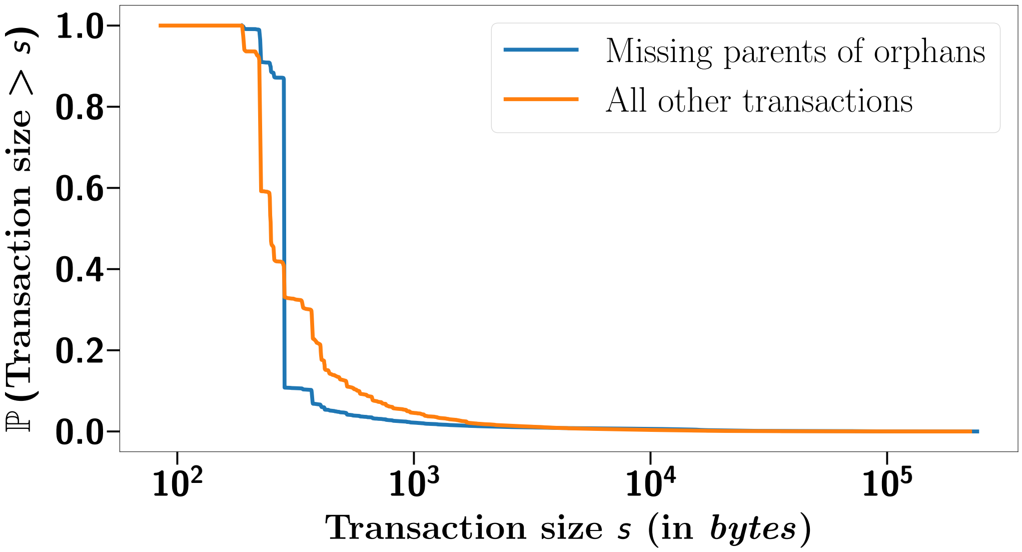


Figure 5·3: CCDFs of transaction size of missing parents of orphan transactions, and transaction size of all other transactions.

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.

5.1.5 Relating transaction fee to size of missing parents

We showed in **Sections** 5.1.3 and 5.1.4 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, **Figure** 5·4 shows the CDF of transaction fee *per byte* (in satoshis)

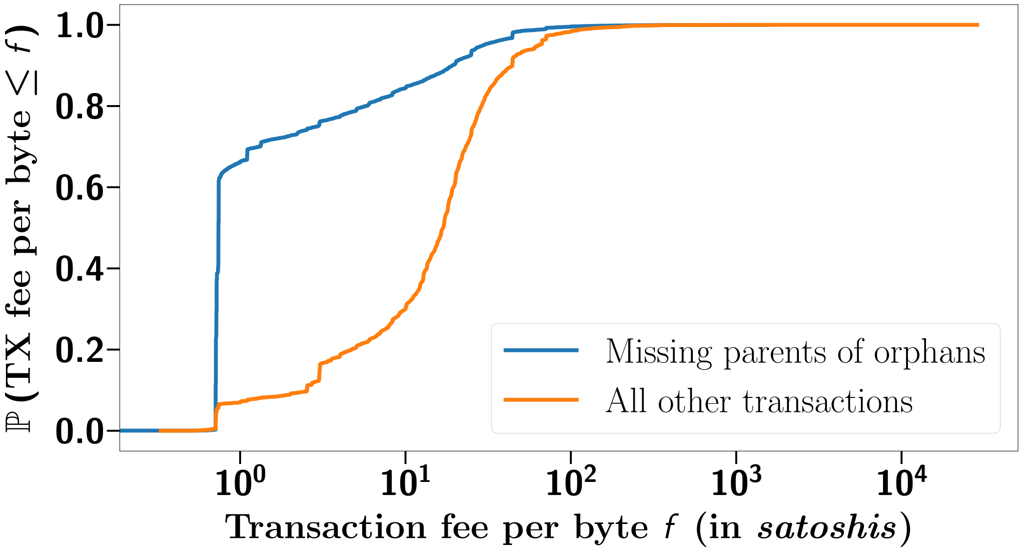


Figure 5·4: CDFs of transaction fee per byte of missing parents of orphan transactions, and transaction fee per byte of all other transactions.

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 [238], possibly because of configurable mempool size [239]. Note that nodes may choose not to accept transactions with a low transaction fee per byte to their mempool, and thereby not propagate them

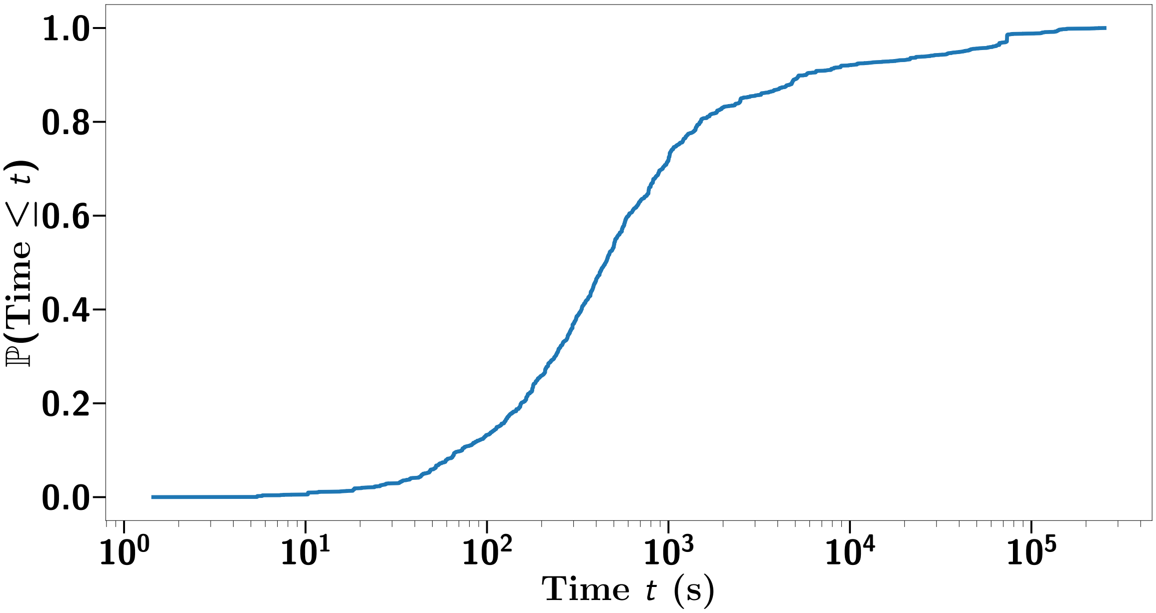


Figure 5·5: CDF of time elapsing from the point a transaction is removed from the orphan pool because its missing parents are found till the block containing the said orphan transaction is received.

further [240].

5.1.6 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 **Section 5.2.3** 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, **Figure 5·5** shows, on an aggregate level, the CDF of the 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. 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 transactions remain in that state until they are added into a block. This could lead to inefficiencies in the Bitcoin protocol [68, 241] since transactions are not propagated to peers for as long as they remain orphan (see **Section 2.3**).

5.1.7 Delay in receiving missing parents from peers

As noted in **Section 2.3**, 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 period (see **Section 5.1.1**) which represents about 15.98% of the total number of missing parents recorded.

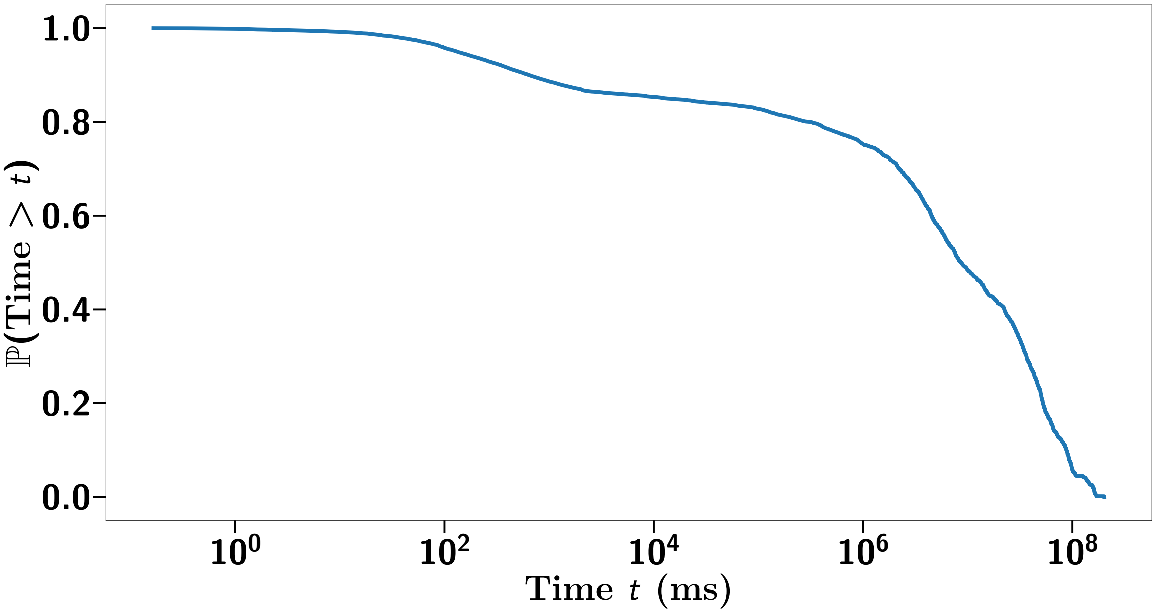


Figure 5·6: CCDF of time elapsing from the point a transaction becomes orphan till one of its parents is found.

Figure 5·6 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.

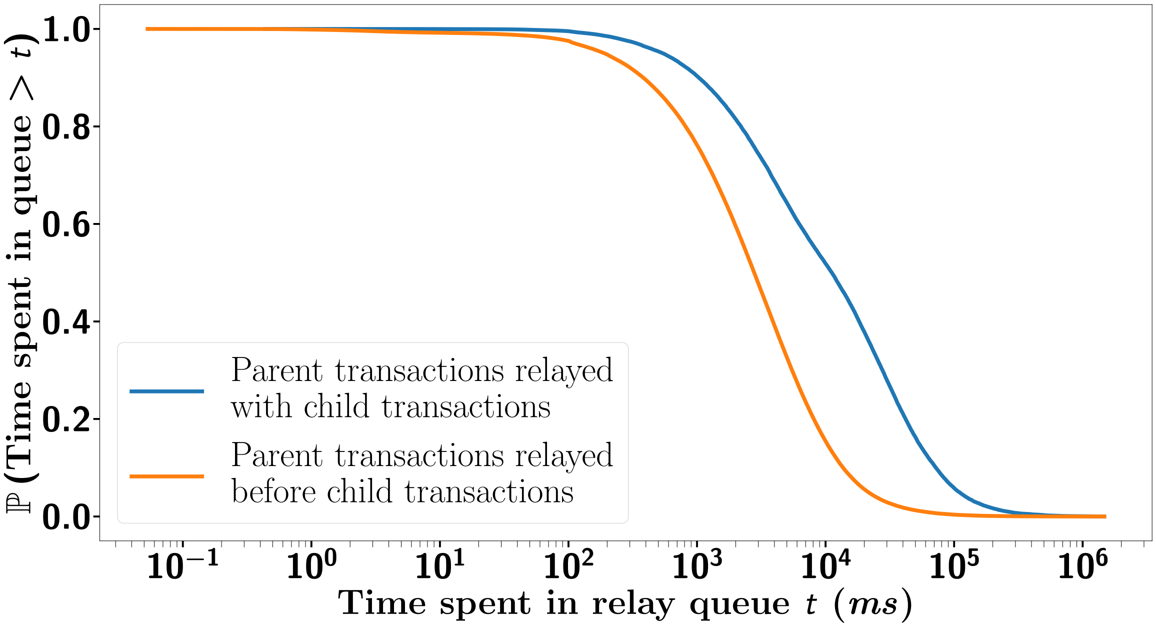


Figure 5·7: CCDFs of time spent in relay queue of parent transactions that are relayed with and parent transactions that are relayed before their respective child transactions.

5.1.8 Impact of transaction fee

The findings in **Section 5.1.6** 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

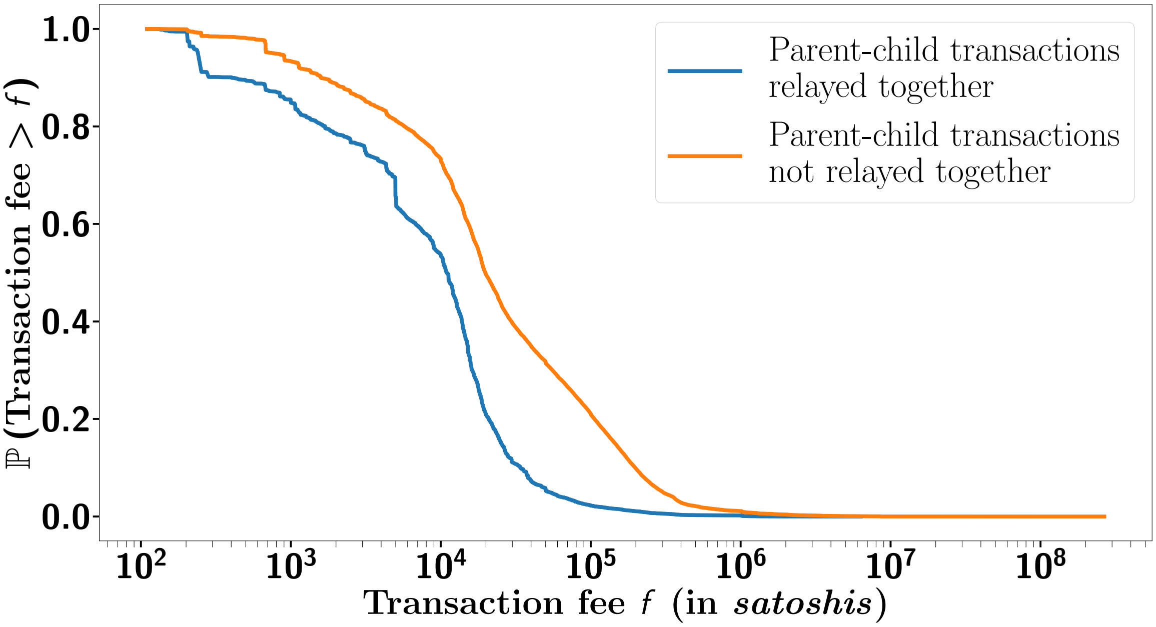


Figure 5·8: CCDFs of transaction fee of parent transactions that are relayed with and parent transactions that are relayed before their respective child transactions.

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.

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. **Figure** 5·7 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 **Figure 5·8**, 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.

5.2 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.

5.2.1 Measurement setup

For **Sections 5.2.2** to **5.2.5**, we extend our measurement setup from **Section 5.1.1** 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 EST to November 25, 2019 10:59 AM EST, 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 EST to December 02, 2019 10:59 AM EST, 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 1,000 transactions (nodes N_5 and N_6). We have made all relevant logs generated during the experiments open source and accessible on GitHub [242].

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.

Figure 5·9 and **Figure 5·10** 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

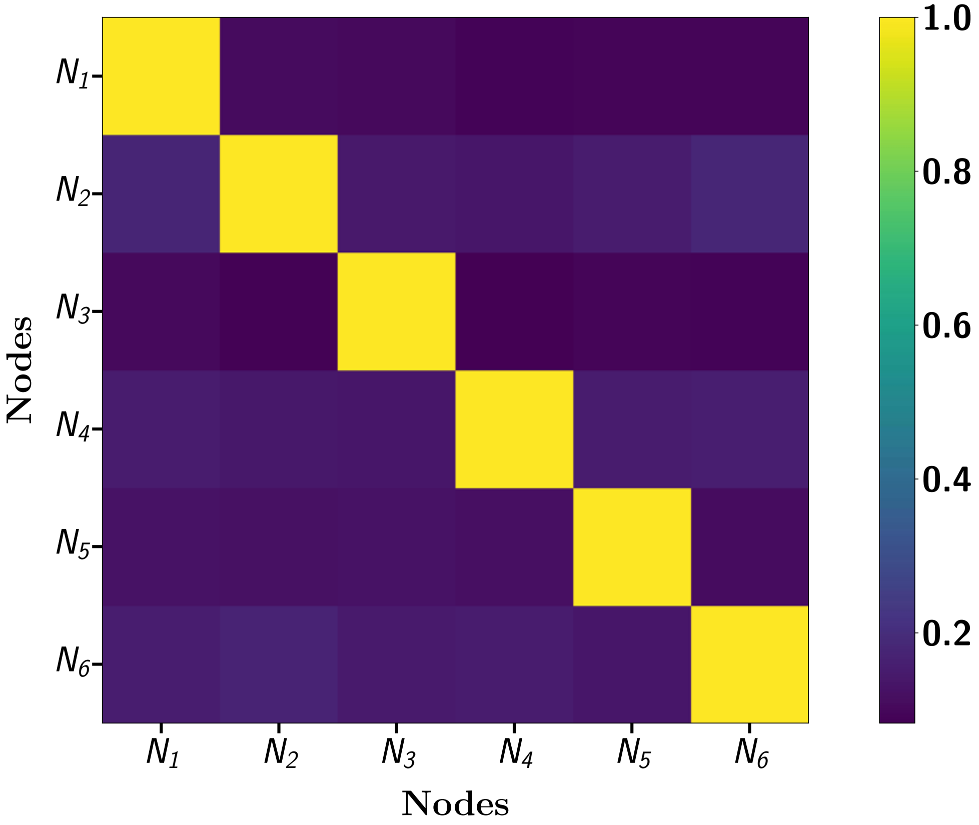


Figure 5·9: Similarity matrix depicting average number of common peers across nodes during the first round of measurement period.

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 **Section** 5.2.6, we extend our measurement setup from **Section** 5.1.1 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 EST to July 1, 2020 11:59 AM EST. We configure two nodes with a timeout of 10 minutes, two nodes with a timeout of 15 minutes, four nodes with the default timeout

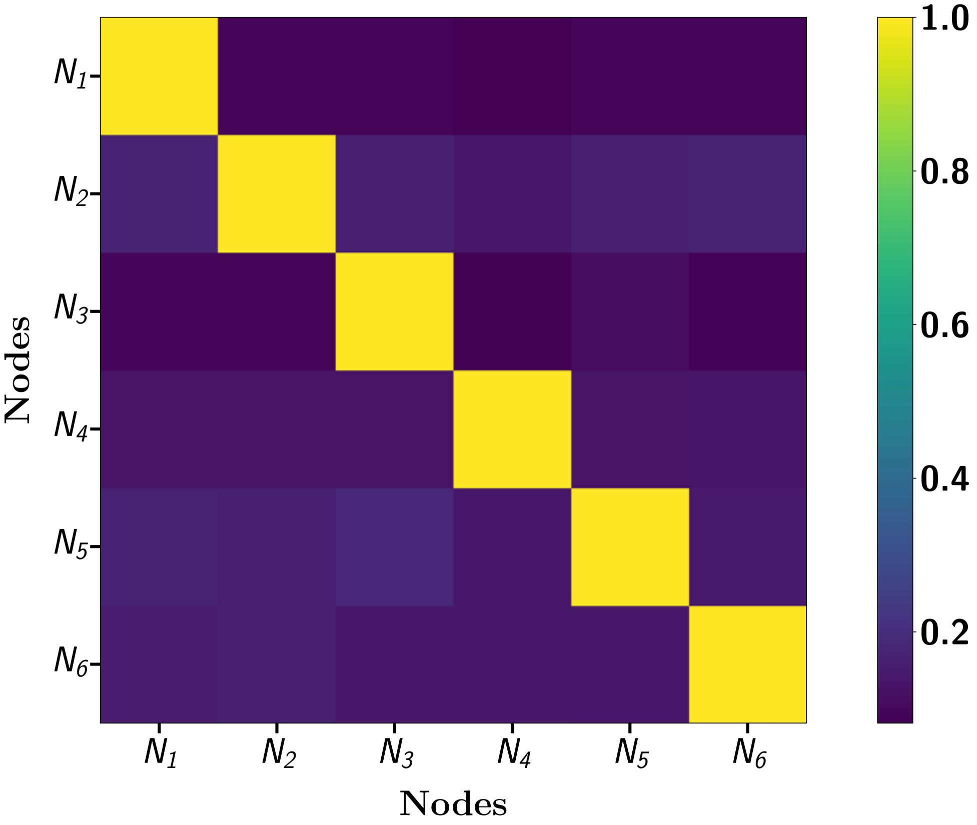


Figure 5·10: Similarity matrix depicting average number of common peers across nodes during the second round of measurement period.

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 transactions, since smaller sizes lead to a large fraction of evictions as discussed in **Section** 5.2.2.

5.2.2 Removal of orphan transactions from orphan pool

As specified in **Section** 2.3, 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, **Figure** 5·11 shows the fraction of transactions removed from the orphan transaction falling within each of the six cases across the nodes with varying

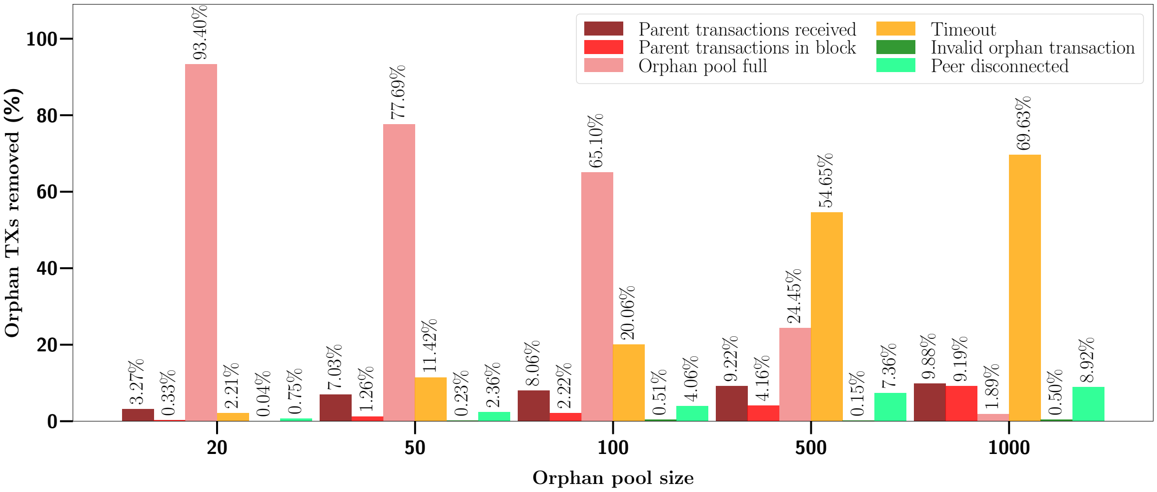


Figure 5·11: Fraction of orphan transactions that are removed from the orphan pool due to each of the six causes across all nodes, under different pool sizes.

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 1,000 (*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, across nodes, is that the node receives the missing parent it had requested from its peers. **Figure** 5·11 shows that as the size of the orphan pool increases, the fraction of orphan transactions that receive their respective missing

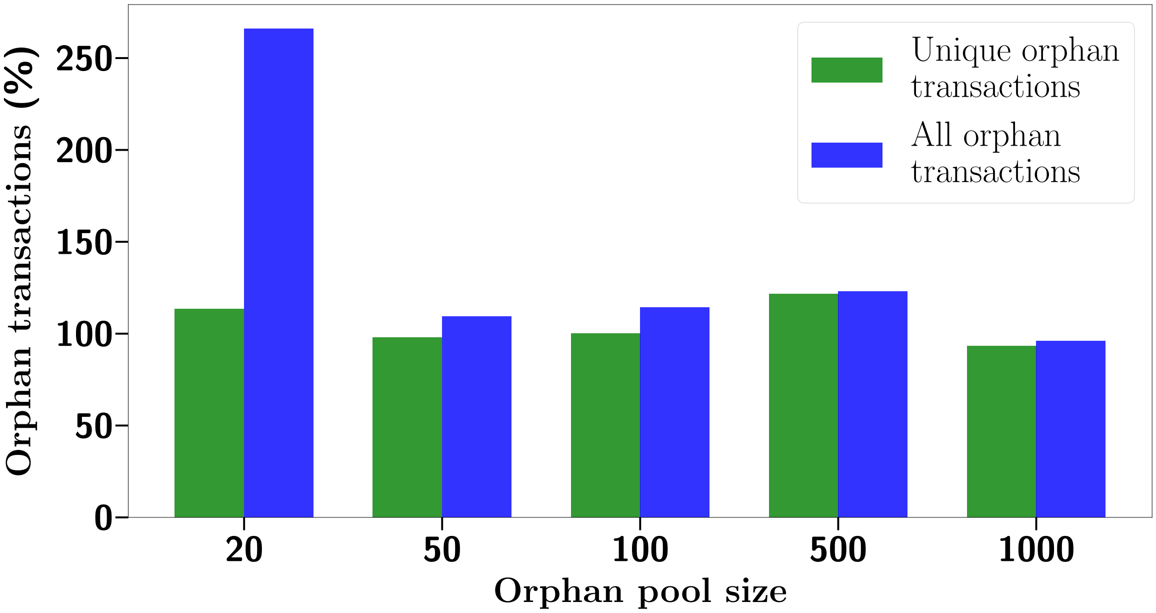


Figure 5·12: Number of unique and total number of orphan transactions received across nodes with varying orphan pool sizes.

parents gradually increases.

5.2.3 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 (see **Section 2.3**), 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 **Figure 5·12** 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.

5.2.4 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 [243] and 32 bytes for the transaction hash in the `getdata` message [244]. 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.

Figure 5·13 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 **Figure 5·13**. On the other hand, for an orphan pool size of 1,000, 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 (see **Section 5.2.3**).

5.2.5 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.

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 5.1** shows the *average* CPU usage of the Bitcoin process over the

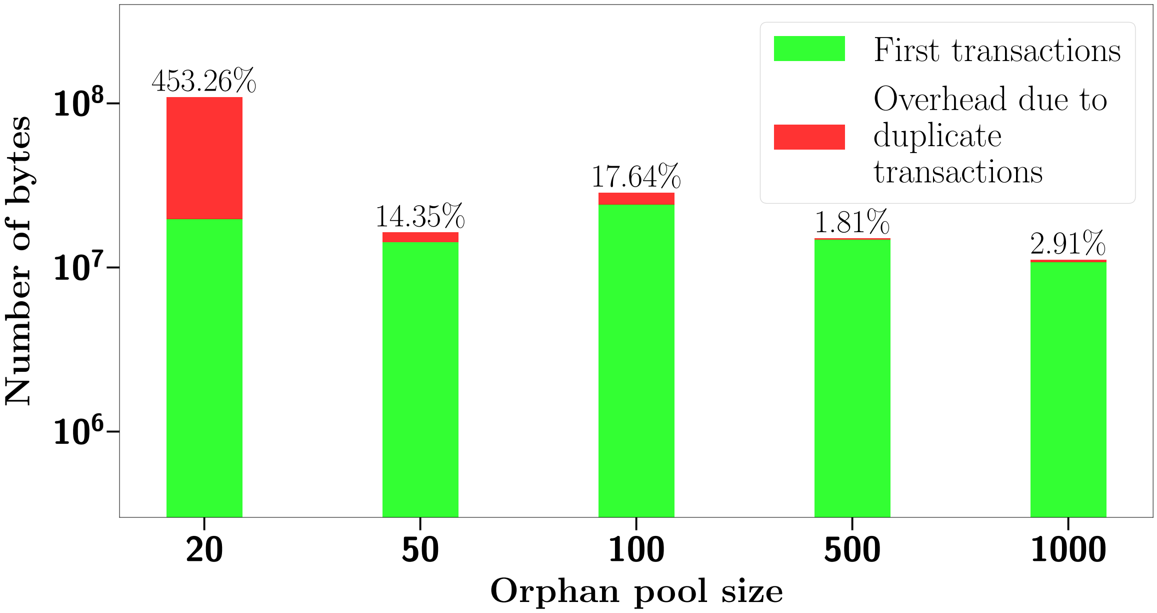


Figure 5·13: Network overhead incurred by nodes with varying orphan pool sizes across nodes.

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 [245–247].

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 [248], (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).

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

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 1,000 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 1,000 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.

5.2.6 Varying orphan transaction timeouts

The findings in **Section 5.2.2** 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.

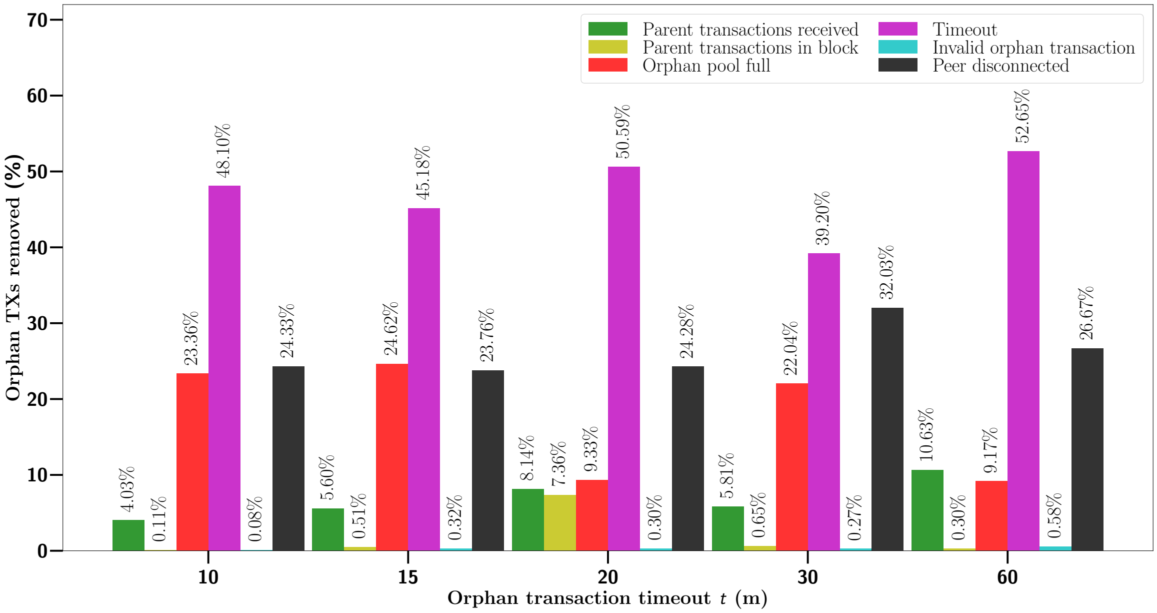


Figure 5·14: Fraction of orphan transactions that are removed from the orphan pool due to each of the six causes across all nodes, under different timeouts.

Figure 5·14 depicts the fraction of transactions removed from the orphan pool for each of the six cases specified in **Section** 2.3 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.

5.3 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.

5.3.1 Measurement setup

We configure three nodes with the same identical hardware and software specifications as described in **Section 5.1.1**. 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 EST for two weeks. That is, we collect and present results obtained from data gathered over 84 sessions that are 12 hours long.

5.3.2 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.

Figure 5·15 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 **Figure 5·16**.

We note that a node has fewer peers when it starts up as compared to in steady-

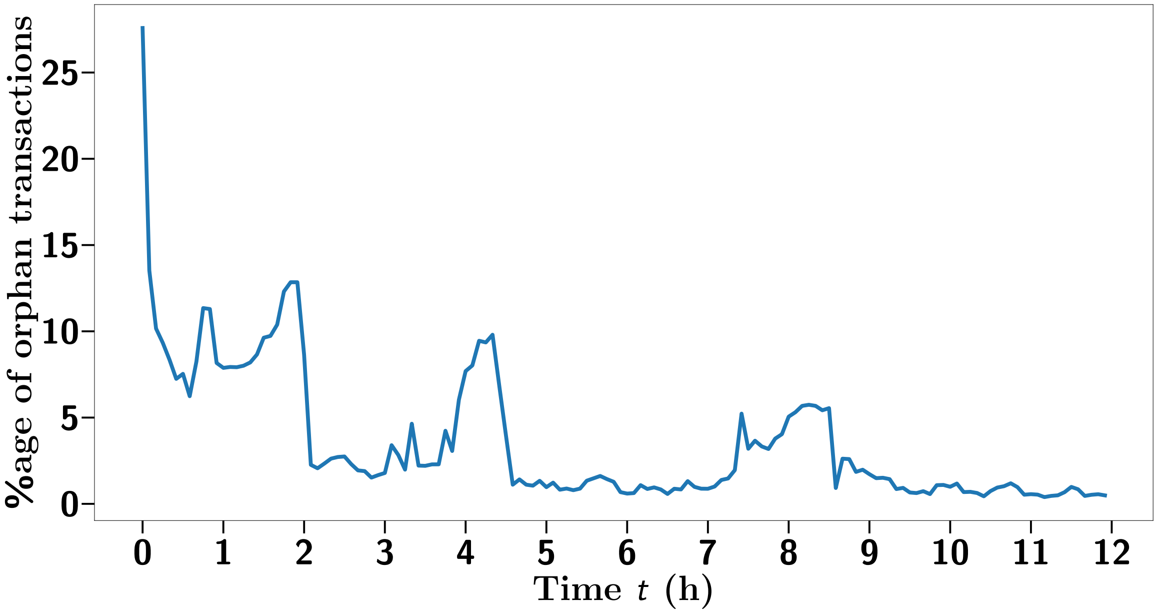


Figure 5·15: Percentage of transactions that become orphan during each 5-minute bin interval of the 12 hour long sessions.

state. Therefore, it receives a relatively smaller number of transactions at the beginning, as shown in **Figure 5·16**.

5.3.3 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.

Figure 5·17 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 hour measurement sessions.

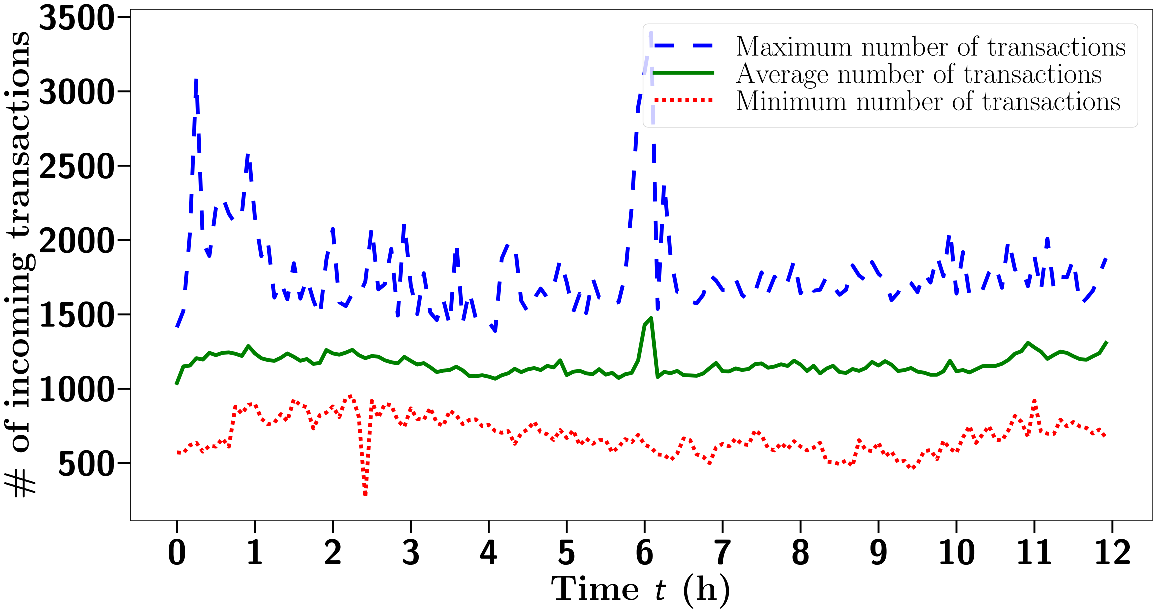


Figure 5·16: Maximum, average, and minimum number of transactions received by nodes during each 5-minute bin interval of the 12 hour long sessions aggregated over all 84 sessions. The differences between the curves indicate that the number of incoming transactions varies across sessions.

5.3.4 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 arrives within the first two hours. Hence, we zoom into this time frame and examine how each of the six scenarios (see **Section 2.3**) contributes to transactions being removed from the orphan pool.

Figure 5·18 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 is removed from the orphan pool because their missing parents are found.

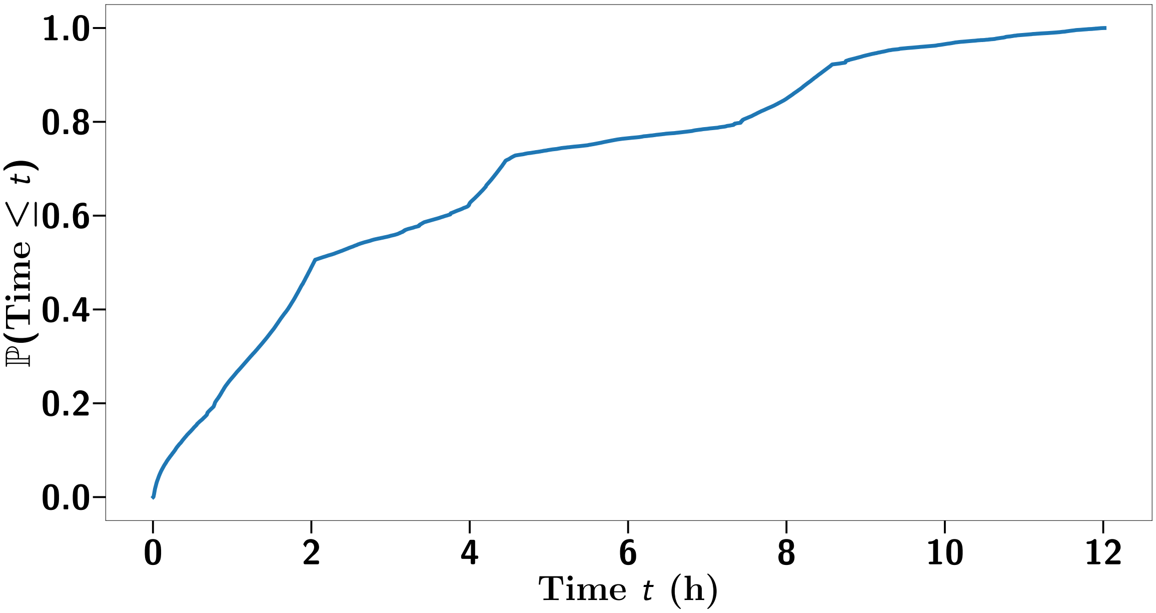


Figure 5·17: CDF of arrival times of orphan transactions during measurement periods. Roughly 50% of all orphan transactions are received in the first two hours.

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 (see **Section 5.2.4**).

5.4 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. **Section 2.3**). 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

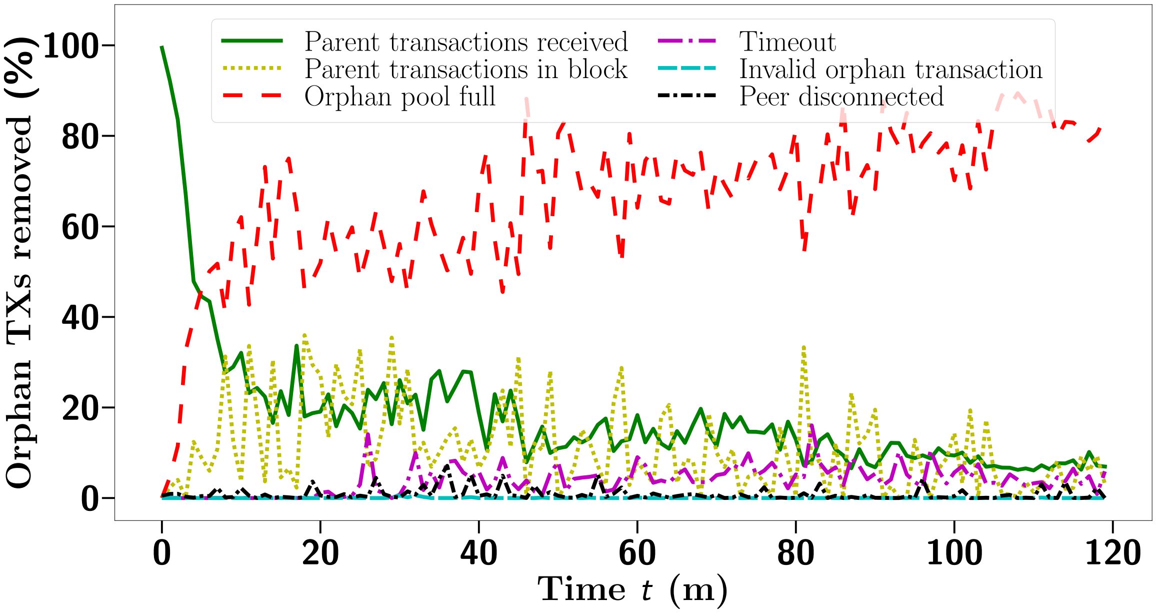


Figure 5·18: Fraction of transactions that are removed from the orphan pool due to each of the six causes (see **Section** 2.3) over the first two hours of the 12 hour long sessions.

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** 5.3 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 [68, 241], 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 5.3** 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 [249–251] 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.

5.5 Summary

The main results from the work presented in this chapter are summarized below.

- We have performed the first ever characterization of orphan transactions in the Bitcoin network. Our analysis shows that contrary to natural conjecture, orphan transactions, on average, have *fewer* parents than non-orphan transactions. The parents of orphan transactions have (*i*) lower fees, (*ii*) larger size, and (*iii*) lower fee-per-byte as compared to parents of non-orphan transactions.

- We discover in an extensive analysis that 44.50% of orphan transactions received by the measurement nodes appear in blocks received by the nodes. However, only 11% of these were recovered before the node received the respective block containing these transactions. In addition, 67.58% of the missing parents appear in the *same* block as their orphan child transaction.

- We find that 50% of missing parents are received by measurements *two hours after* the respective child transaction is added to the orphan pool. Indeed, roughly 35% of missing parents are received after the mean difference of roughly 8 hours.

- We have documented the network and performance overhead incurred by orphan transactions for orphan pools of various sizes, and varying default timeouts. We find that increasing the orphan pool size from the default capacity of 100 to a slightly higher size of 1,000 transactions significantly reduces network overhead while adding negligible performance overhead. On the other hand, increasing the default timeout from the default of 20 minutes does not improve performance whereas reducing the timeout value deteriorates performance.

- We have also studied the transient behavior of orphan transactions in nodes that join the network for the first time or after a long downtime. These nodes do not contain useful transactions in their mempools. We observer that roughly 25% of the transactions received by the measurement nodes become orphan whereas 50% of these transactions are added to the orphan pool within the first two hours of the node (re)joining the network and transactions being evicted from the orphan pool due to it becoming full is the dominant cause for these two hours.

It is apparent from results presented in this chapter that transactions becoming orphan result in delays in their relay to peers of a Bitcoin node. Bitcoin users can utilize the findings from this chapter to appropriately set their own transaction fees for relay of the corresponding transactions in the network. It should be beneficial for full nodes to configure their orphan pools with a size slightly larger than the default to reduce unnecessary network overhead. This should be especially useful for churning nodes who rejoin the network after a long downtime.

Chapter 6

Conclusions and future work

In this thesis, we studied the occurrence and impacts of natural phenomena in popular blockchain systems namely Bitcoin and Bitcoin Cash. We showed that effects of such phenomena can be detrimental to the performance of the blockchain systems and proposed novel solutions to tackle these unfavorable effects.

Our main contributions are as follows: we first devise a framework to help study information related to events that occur in a blockchain system. Next, with the help of this framework, we examined (i) the effects of churn on the relay of blocks in the Bitcoin protocol, (ii) the performance of different block relay protocols implemented in the Bitcoin Cash network under realistic network conditions, (iii) the characteristics of orphan transactions in the Bitcoin network and the overhead imposed by them whilst using the default parameters, and (iv) the impact of churn on transactions received by Bitcoin nodes becoming orphan. Finally, we (i) proposed, implemented, and evaluated a novel synchronization scheme and demonstrated its usefulness in mitigating the detrimental effects of churn on the Bitcoin system, and (ii) showed that the overhead caused by frequent eviction of orphan transactions from the orphan pool can be significantly reduced by slightly increasing the size of the pool.

In the rest of this section, we summarize our results and discuss potential future work directions.

Summary of contributions and findings

We summarize our contributions and findings from this thesis below.

Churn in the Bitcoin Network. In **Chapter 3**, we identified and empirically demonstrated the heretofore undocumented effect of *churn* on the Bitcoin network. We performed a thorough characterization of churn, including the daily churn rate and statistical fitting of the distributions of the lengths of up and down sessions. This statistical characterization should prove useful to other researchers, for the purpose of analyzing, simulating, and emulating the behavior of the Bitcoin network.

We also used the statistical characterization to evaluate the impact of churn on the propagation delay of blocks in the live Bitcoin network. In the process of this research, we developed a logging mechanism for tracing events in Bitcoin nodes, which we have released for public use [121]. Our experiments showed that churn produces a marked degradation in the performance of the delay-optimized compact block protocol. This is because unsuccessful compact blocks are much more prevalent in churning nodes, and the associated incomplete blocks often miss a large number of transactions (78.08 on average). As a result, the propagation delay of blocks processed by churning nodes is substantially larger, on average, than that of nodes that are always connected. In fact, occurrences of propagation delays that exceed one second are common. Our measurements show that more than 6% of the blocks processed by the churning nodes have a propagation delay exceeding one second, compared to less than 1% of the blocks processed by the control nodes. Note that this corresponds to the delay over a single hop on the Bitcoin network, and hence the end-to-end delay would be even larger.

We have also proposed and implemented into Bitcoin Core a proof-of-concept synchronization scheme, `MempoolSync`, that sends transactions to peers in an effort to alleviate the impact of churn and keep mempools of nodes synchronized. Our experimental results show that churning nodes that accept `MempoolSync` messages are able to successfully reconstruct, on average, a larger fraction of compact blocks that they receive as compared to churning nodes that do not accept such messages. This happens because the former miss far fewer transactions (about 3 times less on average) from the compact blocks that they receive. As a result, the churning nodes that accept `MempoolSync` messages experience block propagation delay that is, on average, slightly less than half the propagation delay of churning nodes that do not accept such messages.

Comparison of block relay protocols. We have studied the empirical performance of three popular block relay protocols (Graphene, compact blocks, and the Bitcoin default) on a live blockchain through the Bitcoin Unlimited (BU) client in a variety of network regimes and presented our findings in **Chapter 4**. In our experiments on nodes that are in the `always on` and `statistical churn` regimes, the Graphene block relay protocol performed best and the Bitcoin default block relay protocol performed the worst in terms of average block communication sizes and propagation delays: compared to Graphene, compact and default blocks have roughly 40% and 500% higher propagation delays, and over 80% and 150% larger communication sizes. As a result, it seems preferable to configure nodes with the Graphene block relay protocol under typical network conditions.

We have also studied the effects of periodic churn on the performance of the block relay protocols. We ran two sets of experiments (at 25% and 75% off-duty cycles respectively) on nodes, whose connectivity fluctuated in periods of 20 m, 1 h, 3 h, and 6 h. We found that the default block relay protocol performed worse than Graphene and compact blocks, in terms of propagation delay and communication sizes, regardless of the off duty cycle. Graphene generally performed better than compact blocks in the 25% off-duty cycle, and vice versa in the 75% regime. More precisely, Graphene performance significantly degrades when the destination misses many transactions as this causes additional rounds of communication. As a result, if nodes churn frequently or are off the network for long periods of time, it may be preferable to configure them with the compact block protocol.

We further conducted a temporal analysis of the Graphene block relay to identify which decoding scenario typically prevails. In particular, we found out that scenario ③ of the protocol occurs infrequently. This result suggests that the block failure recovery procedure in the protocol could be further optimized to avoid unnecessary rounds of communication required to recover missing transactions should scenario ④ occur. For example, it may be beneficial to send hashes of transactions when initiating failure recovery instead of encoding them in a Bloom filter so that there is no chance of false positives. Doing so should only add negligible overhead since scenarios including failure recovery occur infrequently.

Finally, we studied the benefit of including additional transactions in the initial message of the compact block relay protocol. These are full transactions predicted by the `SRC` node to be missing from the mempool of the `DST` node. Our analysis shows that in our experimental nodes in both `always on` and `statistical churn` regimes, a large portion of the `cmpctblock` messages only contain the coinbase transaction. Of the remaining portion of the `cmpctblock` messages, over 90% of the additional transactions result in wasted bandwidth. It may, therefore, be interesting to evaluate whether excluding additional transactions from the `cmpctblock` messages has greater benefits.

Orphan transactions in the Bitcoin network. We have investigated circumstances under which a Bitcoin transaction is orphaned in **Chapter 5**. 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 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). 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.

Future work directions

Finally, we discuss potential venues for future research based on the work presented in this thesis.

Churn in different blockchain systems. In this thesis, we have shown the existence and impact of churn on the Bitcoin network. We believe our methodology to accomplish this can be extended to other blockchains to study the effects of churn on the respective systems. Doing so should provide a large enough data set which provides insights into designing churn-tolerant block relay protocols to blockchain developers.

Churn in layer-2 solutions. The problem of scalability is widespread in the blockchain ecosystem. To tackle this challenge, *layer-2* solutions, which are networks or technologies built on top of an underlying blockchain protocol, have been proposed and implemented for different blockchains [252–254]. In the *lightning network* [252], for example, two parties can open an *off-chain* private channel between themselves by locking a certain amount of funds to the Bitcoin network. The parties can then transfer funds between themselves via the private channel without the need to record each transfer to the main blockchain. Their balances are only updated on the main blockchain when they close the private channel between them. Since the parties do not congest the main blockchain with each transfer record, the lightning network can significantly improve the transaction throughput of the Bitcoin network thus improving the scalability of the protocol. A security threat to layer-2 solutions appears when participating parties churn and go offline: they become vulnerable to an adversary since they are no longer synchronized with the payment channel network (PCN) [255] and the blockchain [256]. A study on how churn affects layer-2 solutions and methods to mitigate such effects should be valuable.

Implementation of Graphene in other blockchains. The results of this thesis indicate that the Graphene block relay protocol should be of interest to other blockchains, including BTC, the main Bitcoin blockchain. Integrating and evaluating this protocol within Bitcoin Core, the client software for the Bitcoin protocol, appears to be an effort worth investing.

Extended comparison of block relay protocols. We have presented a comparison of performance of three different block relay protocols in this thesis. It should be interesting to identify if there exist better block relay protocols and how they compare against the ones presented in our work. For example, *Velocity* [61] aims to improve the propagation of blocks through rateless erasure coding. However, as noted in **Chapter 1**, this protocol is only evaluated via simulations. It is also not clear how Velocity performs compared to other block relay protocols such as the Graphene block relay protocol under realistic network conditions.

Blockchain-wide implementation of mempool synchronization. As an outcome of the work presented in this thesis, it is evident that there is significant benefit in implementing efficient synchronization of the mempools of Bitcoin nodes, thus keeping them up-to-date with transactions that they might have missed while being disconnected. Indeed, our analysis of `MempoolSync` in **Section 3.3** shows that churning nodes that synchronize their mempools with highly-connected nodes have fewer block decode failures and smaller increases in block propagation delays. It should, therefore, be beneficial to study how mempool synchronization schemes perform across different blockchains.

Detailed investigation related to orphan transactions. Our findings in this thesis 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 [165] 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 [68, 241]). Detailed investigation of these phenomena represent interesting areas for future work.

Concluding remarks

This thesis makes significant contributions towards facilitating *in situ* measurements in full nodes and discovering the impact of natural phenomena in the Bitcoin and Bitcoin Cash networks through empirical methods, and proposes novel solutions to provably mitigate those impacts. The techniques discussed in the thesis can be employed in other blockchain systems which opens several venues for future research.

Appendix A

Explanation of log-to-file system

We designed and implemented the *log-to-file* system in Bitcoin Core and Bitcoin Unlimited to facilitate empirical measurements of different events in the underlying protocols. In this section, we explain the motivation behind and the challenges faced when designing the system, the design of the system, demonstrate its usage in different scenarios, and show how useful data can be extracted from the respective logs generated. All file paths in this section are relative to the directory at `https://github.com/nislab/bitcoin-releases/tree/wip/src/`.

A.1 Motivation and challenges

To the best of our knowledge, existing tools that facilitate extraction of data from blockchains are either not publicly available, or are limited in scope of what they can measure. For example, such tools cannot measure the impact of phenomena such as churn of full nodes and orphan transactions on the corresponding blockchain system. To this end, we have designed and implemented a measurement tool dubbed as the *log-to-file* system in the Bitcoin Core and Bitcoin Unlimited software for the purpose of data collection *in-situ*.

We studied the Bitcoin Core and Bitcoin Unlimited software implementations and identified events of interest such as reception of blocks and transactions, addition of transactions to the orphan pool, and so on, with the help of the Bitcoin protocol

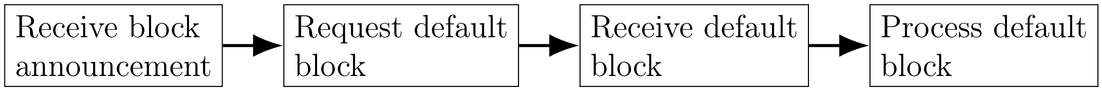


Figure A·1: State transitions for the default block relay protocol from the perspective of a receiver.

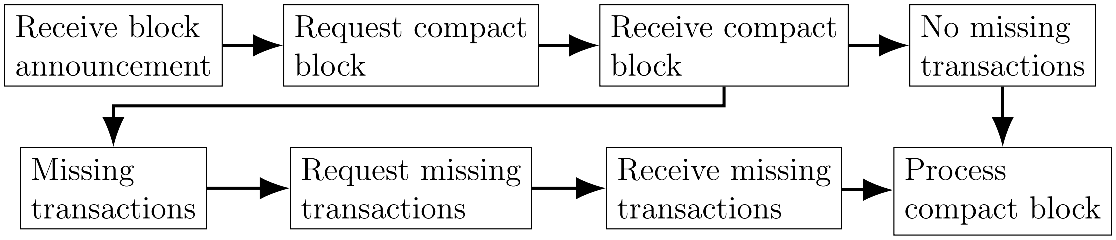


Figure A·2: State transitions for the compact block relay protocol from the perspective of a receiver.

documentation [181]. This process requires a significant effort in itself. To put it into perspective, recall the default block relay protocol (explained in more detail in **Section 2.2.4**) which is quite simple. A peer of the Bitcoin node announces the block which the node requests and upon receiving the block, processes it and relays it to other peers. The transitions in these states are illustrated in **Figure A·1**. The compact block relay protocol (explained in more detail in **Section 2.2.4**) is slightly more complex than the default block relay protocol as illustrated in **Figure A·2**. Now there can also be transactions missing from the mempool of the node when a compact block is received and the node would need to request missing transactions to be able to reconstruct the block and relay it to peers. The Graphene block relay protocol (explained in more detail in **Section 2.2.4**) is much more complex since there can now be IBLT decode failures in addition to missing transactions. A node may need to perform failure recovery and even then can have transactions missing from its mempool. **Figure A·3** illustrates the complexity of the protocol. The process is similar for orphan transactions.

Once an event of interest is identified, we insert hooks into the corresponding place in the software via the methods exposed by the log-to-file system (explained in more

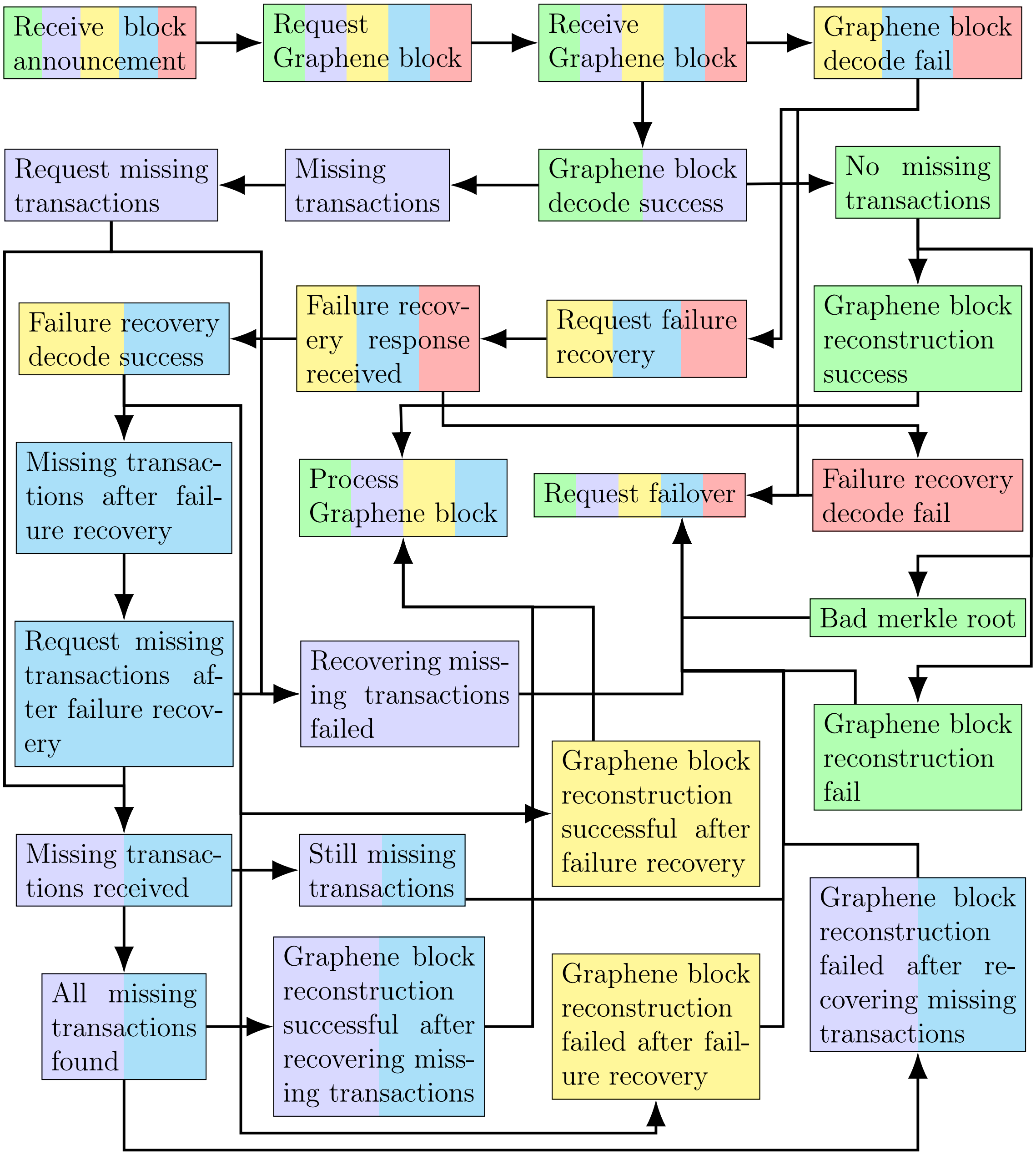

Receive block announcement
Request Graphene block
Receive Graphene block
Graphene block decode fail
Request missing transactions
Missing transactions
Graphene block decode success
No missing transactions
Failure recovery decode success
Failure recovery response received
Request failure recovery
Graphene block reconstruction success
Missing transactions after failure recovery
Process Graphene block
Request failover
Failure recovery decode fail
Bad merkle root
Request missing transactions after failure recovery
Recovering missing transactions failed
Graphene block reconstruction fail
Graphene block reconstruction successful after failure recovery
Missing transactions received
Still missing transactions
Graphene block reconstruction failed after recovering missing transactions
All missing transactions found
Graphene block reconstruction successful after recovering missing transactions
Graphene block reconstruction failed after failure recovery

Figure A·3: State transitions for the Graphene block relay protocol from the perspective of a receiver. States with backgrounds in green, purple, yellow, cyan, and red occur in scenarios ①, ②, ③, ④, and ⑤, respectively (see **Listing** 3). A state with multiple background colors represents more than one corresponding scenarios that transition through it.

detail in **Appendix A.2**). The hooks are triggered when an event of interest occurs and relevant data is recorded to files which can then be post processed to obtain results and insights.

A.2 Design of the system

Similar to Bitcoin Core and Bitcoin Unlimited, the current version of the log-to-file system is implemented entirely in the C++ programming language though it can be ported quite easily to other languages. The system primarily consists of a header file, *i.e.*, `logFile.h`, and a source file, *i.e.*, `logFile.cpp`. The header file exposes the `logFile(...)` method which is overloaded to cover various scenarios along with several other utility functions. The source file provides an implementation of these functions which can be modified based on the users' needs. We demonstrate the usage of some of these functions in **Appendix A.3**.

The logging capability is enabled by adding the `logFile.*` files to the `make` system of Bitcoin Core and Bitcoin Unlimited. Specifically, `logFile.h` must be added to the `BITCOIN_CORE_H` variable[1,2] and `logFile.cpp` must be added to the `libbitcoin_server_a_SOURCES`[3,4] variable in `Makefile.am`[5]. Doing so compiles the source file along with the rest of the Bitcoin software and creates a binary (*i.e.*, `bitcoind` or `bitcoin-qt`) which creates the log files when executed.

A.3 Usage of the system

Before any logs can be generated, the log-to-file system must be initialized using the `initLogger` function[6] which creates the directory structure required to store relevant log files. For example, this can be done when the Bitcoin software itself is

---

[1] `Makefile.am#L112`

[2] `Makefile.am#L255`

[3] `Makefile.am#L267`

[4] `Makefile.am#L344`

[5] `Makefile.am`

[6] `logFile.h#L80`

initializing by placing a call to the function in `init.cpp`[7]. Once the log-to-file system is successfully initialized, it can now be used to log information to files. This can be achieved by including the header file, *i.e.*, `#include <logFile.h>`, in appropriate source files where logging is required.

We now show illustrations for some events for which we generate logs. For the sake of privacy, we replace all IP addresses found in our log file with `aaa.bbb.ccc.ddd`.

Block transfer. The first step in the relay of a block is the announcement of its header from a peer of the measurement node. This announcement is logged by adding the following line to `net_processing.cpp`[8]:

```
logFile("BLCKHEADERRECV -- block header " + pindex->GetBlockHash().
    ToString() + " received from " + pfrom->GetLogName());
```

A record is appended to the log file which contains (*i*) both a human-readable and Unix timestamp at which the header announcement is received, (*ii*) the hash of the block for which the header is announced, and (*iii*) the IP address of the peer who sent the announcement as illustrated below. Note that the node may log several records for the announcement of the same block from different peers since it may be connected to many of them.

```
Wed Nov 10 23:10:40 2021 1636603840938450302 : BLCKHEADERRECV --
    block header 0000000000000000037
    b7fbb0900936afa0be70982637bbff240f840230b765d received from aaa.
    bbb.ccc.ddd:8333 (27)
```

Next, the node checks whether it already knows of this block. In the case that it does not, it sends a request for the block depending on the block relay protocol supported by the node. We only show an illustration for the Graphene block. The request

---

[7] `init.cpp#L1712`

[8] `net_processing.cpp#L1565`

sent to the peer is logged by adding the following line to `requestManager.cpp`[9]:

```
logFile("GRPHNBLCKREQSENT -- graphene block " + obj.hash.ToString()
    + " request of size " + std::to_string(::GetSerializeSize(ss,
    SER_NETWORK, PROTOCOL_VERSION)) + " (bytes), mempool size " + std
    ::to_string(receiverMemPoolInfo.nTx) + " txs sent to peer " +
    pfrom->GetLogName());
```

The record appended to the log file contains (*i*) the hash of the block being requested from the peer, (*ii*) the size (in bytes) of the request sent to the peer, (*iii*) the number of transactions in the mempool of the node which is conveyed to the peer (see **Section 2.2.4**), and (*iv*) the IP address of the peer to whom the request is sent. An illustration of the record is shown below.

```
Wed Nov 10 23:10:41 2021 1636603841049165810 : GRPHNBLCKREQSENT --
    graphene block 0000000000000000037
    b7fbb0900936afa0be70982637bbff240f840230b765d request of size 44
    (bytes), mempool size 1232 txs sent to peer aaa.bbb.ccc.ddd:8333
    (27)
```

The peer now sends the block back to the measurement node. Information about this block is logged by adding the following line to `graphene.cpp`[10]:

```
logFile("GRPHNBLCKRECV -- received graphene block " + pblock->
    grapheneblock->header.GetHash().ToString() + " of size " + std::
    to_string(pblock->grapheneblock->GetSize()) + " (bytes), BFPR = "
     + std::to_string(pblock->grapheneblock->fpr) + ", a = " + std::
    to_string(pblock->grapheneblock->pGrapheneSet->GetIblt()->
    GetHashTableSize()) + " from " + pfrom->GetLogName());
```

The record associated with this information contains (*i*) the hash of the block received from the peer, (*ii*) the size (in bytes) of the received block, (*iii*) some information about the data structures that make up the block, such as the Bloom

---

[9] `requestManager.cpp#L577`

[10] `blockrelay/graphene.cpp#L619`

filter false positive rate, and the number of cells in the IBLT (see **Section 2.2.3**), and (*iv*) the IP address of the peer who sent the block as illustrated below.

```
Wed Nov 10 23:10:41 2021 1636603841079562036 : GRPHNBLCKRECV --
    received graphene block 0000000000000000037
    b7fbb0900936afa0be70982637bbff240f840230b765d of size 5056 (bytes
    ), BFPR = 0.050542, a = 264 from aaa.bbb.ccc.ddd:8333 (27)
```

In addition, the state of the mempool is dumped when a block is received for further analysis by adding the following line to `blockrelay/graphene.cpp`[11]:

```
logFile("mempool", pblock->grapheneblock->header.GetHash().ToString
    ());
```

Hashes of transactions in the mempool are dumped to a file associated with the hash of the block. A truncated illustration of the file is shown below.

```
                                ⋮
01a879cfffe354ad5299161b872605fab70e3d1c1eec9a75a4514cdbf9e8f4d5
0205982a800558009d14aa5fba9b8d2deed386624db6d9be6c0dffe98ae785ed
0208878bdc9fbf1ad8d29b444bd5b6b790c3993a0fd1b11494c00ef5727f50a7
02205070ca32d5983e041adbe1e5754fd07bd0ca5fd5b05479436254c30e4b41
0222c36fdc3bc03fb42c5334541502065e7f3fe8fabfb87695e51fc9903d049b
                                ⋮
```

Block decode. Next, the node attempts to decode the block upon receiving it which can either succeed or fail. Failure in decoding of the Graphene block is recorded by adding the following line in `blockrelay/graphene.cpp`[12]:

```
logFile("GRPHNDECODEFAIL -- graphene decode failed; starting failure
     recovery: " + pblock->grapheneblock->header.GetHash().ToString()
     + " from " + pfrom->GetLogName());
```

---

[11] `blockrelay/graphene.cpp#L620`

[12] `blockrelay/graphene.cpp#L705`

In addition, the cause for the block decode is also logged by adding the following lines in `blockrelay/graphene.cpp`[13]:

```
logFile("GRPHNBLCKIBLTPEELFAIL -- IBLT peeling failed for block " +
    pblock->grapheneblock->header.GetHash().ToString() + " from " +
    pfrom->GetLogName());
```

and[14]:

```
logFile("GRPHNBLCKIBLTRECONFAIL -- " + std::string(e.what()) + " for
     block " + pblock->grapheneblock->header.GetHash().ToString() + "
     from " + pfrom->GetLogName());
```

Appropriate records containing the hash of the block and the cause of decode failure are then appended to the log files as illustrated below.

```
Fri Nov 12 02:44:06 2021 1636703046550497610 : GRPHNDECODEFAIL --
    graphene decode failed; starting failure recovery:
    000000000000000000017ec62b690ceeee1cac9623aa6b49aa4c724bb72dc47fb
    from aaa.bbb.ccc.ddd:50090 (1957)
Fri Nov 12 02:44:06 2021 1636703046550656520 : GRPHNBLCKIBLTPEELFAIL
     -- IBLT peeling failed for block 00000000000000000017
    ec62b690ceeee1cac9623aa6b49aa4c724bb72dc47fb from aaa.bbb.ccc.ddd
    :50090 (1957)
```

Similarly, success in decoding of the Graphene block is recorded by adding the following line in `blockrelay/graphene.cpp`[15]:

```
logFile("GRPHNDECODESCCS -- graphene decode successful: " + pblock->
    grapheneblock->header.GetHash().ToString() + " from " + pfrom->
    GetLogName());
```

This appends a record to the log file containing the hash of the block which is successfully decoded as illustrated below.

---

[13] `blockrelay/graphene.cpp#L707`

[14] `blockrelay/graphene.cpp#L709`

[15] `blockrelay/graphene.cpp#L725`

```
Wed Nov 10 23:10:41 2021 1636603841085376365 : GRPHNDECODESCCS --
    graphene decode successful: 0000000000000000037
    b7fbb0900936afa0be70982637bbff240f840230b765d from aaa.bbb.ccc.
    ddd:8333 (27)
```

Block reconstruction. Once the block is successfully decoded, the node attempts to reconstruct the block the result of which is logged. For example, successful block reconstruction is recorded by adding the following line in `blockrelay/graphene.cpp`[16]:

```
logFile("GRPHNBLCKRECONSCCS -- graphene block reconstruction success
    : " + pblock->grapheneblock->header.GetHash().ToString() + " from
     " + pfrom->GetLogName());
```

This appends a record to the log file which contains the hash of the successfully reconstructed block as illustrated below.

```
Wed Nov 10 23:10:41 2021 1636603841034258350 : GRPHNBLCKRECONSCCS --
     grahene block reconstruction success: 0000000000000000037
    b7fbb0900936afa0be70982637bbff240f840230b765d from aaa.bbb.ccc.
    ddd:8333 (4)
```

The block is then processed which is logged by adding the following line to `parallel.cpp`[17]:

```
logFile("GRPHNBLCKRECONFIN -- processed block " + inv.hash.ToString
    () + " from " + pfrom->GetLogName());
```

A record is appended to the log file including (i) the timestamp when block processing is finished, and (ii) the hash of the block that is processed as illustrated below.

---

[16] blockrelay/graphene.cpp#L744

[17] parallel.cpp#L643

```
Wed Nov 10 23:10:41 2021 1636603841074956879 : GRPHNBLCKRECONFIN --
    processed block 0000000000000000037
    b7fbb0900936afa0be70982637bbff240f840230b765d from aaa.bbb.ccc.
    ddd:8333 (4)
```

Furthermore, hashes of all transactions in the processed block are dumped to a file for further analysis by adding the following line to `parallel.cpp`[18]:

```
logFile(pblock->vtx, inv.hash.ToString(), pfrom->GetLogName(),
    BlockType::GRAPHENE);
```

A file named after the hash of the block is created containing the hashes of the transactions in the block as illustrated below.

```
8e4793a92212b89a2a4ac2d17bd24d7d897a220e427158e2d0013afe9f764ec5
0013175c7bc284a841905029b33dd4731ae83ed290ab72fd913533a473296c94
003514d864862bdf756934cb03ff20c87431249714cd9b520d4445320647563b
0054e6396911d7d3f4c13b8fc929e308599763c9b72a3db8ba081945b96bdd9a
00fb4c991bec86ea93e98e06e6ed9bf335e0891453ea88fe58a9a06cc7983eb9
                                ⋮
```

---

[18] `parallel.cpp#L644`

Bibliography

[1] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system." https://bitcoin.org/bitcoin.pdf, 2008. Online; Accessed: Nov 27, 2021.

[2] V. Buterin *et al.*, "A next-generation smart contract and decentralized application platform." https://blockchainlab.com/pdf/Ethereum_white_paper-a_next_generation_smart_contract_and_decentralized_application_platform-vitalik-buterin.pdf, 2014. Online; Accessed: Nov 27, 2021.

[3] V. Buterin *et al.*, "Ethereum Whitepaper." https://ethereum.org/en/whitepaper/. Online; Accessed: Nov 9, 2021.

[4] "Why Cardano." https://why.cardano.org/. Online; Accessed: Nov 9, 2021.

[5] "The Future of CBDCs: Why All Central Banks Must Take Action." https://ripple.com/wp-content/uploads/2021/01/cbdc-whitepaper-2020.pdf. Online; Accessed: Nov 9, 2021.

[6] "Tether: Fiat currencies on the Bitcoin blockchain." https://tether.to/wp-content/uploads/2016/06/TetherWhitePaper.pdf. Online; Accessed: Nov 9, 2021.

[7] S. Saberi, M. Kouhizadeh, J. Sarkis, and L. Shen, "Blockchain technology and its relationships to sustainable supply chain management," *International Journal of Production Research*, vol. 57, no. 7, pp. 2117–2135, 2019.

[8] R. Casado-Vara, J. Prieto, F. D. la Prieta, and J. M. Corchado, "How blockchain improves the supply chain: case study alimentary supply chain," *Procedia Computer Science*, vol. 134, pp. 393–398, 2018. The 15th International Conference on Mobile Systems and Pervasive Computing (MobiSPC 2018) / The 13th International Conference on Future Networks and Communications (FNC-2018) / Affiliated Workshops.

[9] S. A. Abeyratne and R. Monfared, "Blockchain ready manufacturing supply chain using distributed ledger," *International Journal of Research in Engineering and Technology*, vol. 05, no. 09, pp. 1–10, 2016.

[10] K. Francisco and D. Swanson, "The Supply Chain Has No Clothes: Technology Adoption of Blockchain for Supply Chain Transparency," *Logistics*, vol. 2, no. 1, 2018.

[11] M. Mettler, “Blockchain technology in healthcare: The revolution starts here,” in *2016 IEEE 18th International Conference on e-Health Networking, Applications and Services (Healthcom)*, pp. 1–3, 2016.

[12] C. C. Agbo, Q. H. Mahmoud, and J. M. Eklund, “Blockchain Technology in Healthcare: A Systematic Review,” *Healthcare*, vol. 7, no. 2, 2019.

[13] E. J. De Aguiar, B. S. Faiçal, B. Krishnamachari, and J. Ueyama, “A Survey of Blockchain-Based Strategies for Healthcare,” *ACM Computing Surveys*, vol. 53, Mar. 2020.

[14] W. J. Gordon and C. Catalini, “Blockchain Technology for Healthcare: Facilitating the Transition to Patient-Driven Interoperability,” *Computational and Structural Biotechnology Journal*, vol. 16, pp. 224–230, 2018.

[15] A. Reyna, C. Martín, J. Chen, E. Soler, and M. Díaz, “On blockchain and its integration with IoT. Challenges and opportunities,” *Future Generation Computer Systems*, vol. 88, pp. 173–190, 2018.

[16] X. Fan and Q. Chai, “Roll-DPoS: A Randomized Delegated Proof of Stake Scheme for Scalable Blockchain-Based Internet of Things Systems,” in *Proceedings of the 15th EAI International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services*, MobiQuitous ’18, (New York, NY, USA), p. 482–484, Association for Computing Machinery, 2018.

[17] S. Huh, S. Cho, and S. Kim, “Managing IoT devices using blockchain platform,” in *2017 19th International Conference on Advanced Communication Technology (ICACT)*, pp. 464–467, 2017.

[18] A. Dorri, S. S. Kanhere, and R. Jurdak, “Towards an Optimized BlockChain for IoT,” in *2017 IEEE/ACM Second International Conference on Internet-of-Things Design and Implementation (IoTDI)*, pp. 173–178, 2017.

[19] M. Samaniego, U. Jamsrandorj, and R. Deters, “Blockchain as a Service for IoT,” in *2016 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, pp. 433–436, 2016.

[20] O. Novo, “Blockchain Meets IoT: An Architecture for Scalable Access Management in IoT,” *IEEE Internet of Things Journal*, vol. 5, no. 2, pp. 1184–1195, 2018.

[21] A. A. Monrat, O. Schelén, and K. Andersson, “A Survey of Blockchain From the Perspectives of Applications, Challenges, and Opportunities,” *IEEE Access*, vol. 7, pp. 117134–117151, 2019.

[22] U. Srinivas Aditya, R. Singh, P. K. Singh, and A. Kalla, “A Survey on Blockchain in Robotics: Issues, Opportunities, Challenges and Future Directions,” *Journal of Network and Computer Applications*, vol. 196, p. 103245, 2021.

[23] T. Hewa, M. Ylianttila, and M. Liyanage, “Survey on blockchain based smart contracts: Applications, opportunities and challenges,” *Journal of Network and Computer Applications*, vol. 177, p. 102857, 2021.

[24] D. Di Francesco Maesa and P. Mori, “Blockchain 3.0 applications survey,” *Journal of Parallel and Distributed Computing*, vol. 138, pp. 99–114, 2020.

[25] F. Antonucci, S. Figorilli, C. Costa, F. Pallottino, L. Raso, and P. Menesatti, “A review on blockchain applications in the agri-food sector,” *Journal of the Science of Food and Agriculture*, vol. 99, no. 14, pp. 6129–6138, 2019.

[26] M. Pincheira, M. Vecchio, R. Giaffreda, and S. S. Kanhere, “Exploiting constrained IoT devices in a trustless blockchain-based water management system,” in *2020 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, pp. 1–7, 2020.

[27] N. Bore, A. Kinai, P. Waweru, I. Wambugu, J. Mutahi, E. Kemunto, R. Bryant, and K. Weldemariam, “AGWS: Blockchain-enabled Small-scale Farm Digitization,” in *2020 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, pp. 1–9, 2020.

[28] J. Abou Jaoude and R. George Saade, “Blockchain Applications – Usage in Different Domains,” *IEEE Access*, vol. 7, pp. 45360–45381, 2019.

[29] “Bitcoin’s kryptonite: The 51% attack..” `https://bitcointalk.org/index.php?topic=12435.0`. Online; Accessed: Aug 24, 2020.

[30] S. M. H. Bamakan, A. Motavali, and A. Babaei Bondarti, “A survey of blockchain consensus algorithms performance evaluation criteria,” *Expert Systems with Applications*, vol. 154, p. 113385, 2020.

[31] A. Dorri, S. S. Kanhere, R. Jurdak, and P. Gauravaram, “LSB: A Lightweight Scalable Blockchain for IoT security and anonymity,” *Journal of Parallel and Distributed Computing*, vol. 134, pp. 180–197, 2019.

[32] "Global Cryptocurrency Adoption Doubled Since Jan." https://blog.crypto.com/global-crypto-users-over-200-million/, Jul 2021. Online; Accessed: Dec 20, 2021.

[33] R. Branson. https://twitter.com/richardbranson/status/403884094397759489?s=20, Nov 2013. Online; Accessed: Dec 20, 2021.

[34] "AT&T is the First Mobile Carrier to Accept Payment in Cryptocurrency." https://about.att.com/story/2019/att_bitpay.html, May 2019. Online; Accessed: Dec 20, 2021.

[35] "BITCOIN ACCEPTED." https://promotions.newegg.com/nepro/16-6277/index.html. Online; Accessed: Dec 20, 2021.

[36] CoinMarketCap, "Bitcoin price today, BTC live marketcap, chart, and info." https://coinmarketcap.com/currencies/bitcoin/. Online; Accessed: Nov 9, 2021.

[37] CoinMarketCap, "Global cryptocurrency market charts." https://coinmarketcap.com/charts/. Online; Accessed: Nov 10, 2021.

[38] 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)*, pp. 1–9, 2020.

[39] M. A. Imtiaz, D. Starobinski, and A. Trachtenberg, "Investigating Orphan Transactions in the Bitcoin Network," *IEEE Transactions on Network and Service Management*, vol. 18, no. 2, pp. 1718–1731, 2021.

[40] 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.

[41] 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)*, (Washington, D.C.), USENIX Association, Aug. 2015.

[42] "CVE-2012-3789." https://en.bitcoin.it/wiki/CVE-2012-3789. Online; Accessed: Feb 12, 2020.

[43] "What are orphaned and stale blocks?." https://bitcoin.stackexchange.com/q/5859, Dec 2012. Online; Accessed: Nov 24, 2021.

[44] J. Göbel and A. Krzesinski, “Increased block size and Bitcoin blockchain dynamics,” in *2017 27th International Telecommunication Networks and Applications Conference (ITNAC)*, pp. 1–6, 2017.

[45] C. Decker and R. Wattenhofer, “Information propagation in the bitcoin network,” in *13th IEEE International Conference on Peer-to-Peer Computing (IEEE P2P 2013)*, pp. 1–10, IEEE, 2013.

[46] I. Eyal and E. G. Sirer, “Majority is Not Enough: Bitcoin Mining is Vulnerable,” *Communications of the ACM*, vol. 61, p. 95–102, June 2018.

[47] K. Croman, C. Decker, I. Eyal, A. E. Gencer, A. Juels, A. Kosba, A. Miller, P. Saxena, E. Shi, E. Gün Sirer, D. Song, and R. Wattenhofer, “On Scaling Decentralized Blockchains,” in *Financial Cryptography and Data Security* (J. Clark, S. Meiklejohn, P. Y. Ryan, D. Wallach, M. Brenner, and K. Rohloff, eds.), (Berlin, Heidelberg), pp. 106–125, Springer Berlin Heidelberg, 2016.

[48] S. Rahmadika, S. Noh, K. Lee, B. J. Kweka, and K.-H. Rhee, “The dilemma of parameterizing propagation time in blockchain P2P network,” *Journal of Information Processing Systems*, vol. 16, no. 3, pp. 699–717, 2020.

[49] E. Heilman, A. Kendler, A. Zohar, and S. Goldberg, “Eclipse Attacks on Bitcoin’s Peer-to-Peer Network,” in *24th USENIX Security Symposium (USENIX Security 15)*, (Washington, D.C.), pp. 129–144, USENIX Association, Aug. 2015.

[50] J. Garzik, T. Harding, and D. V. Johannsson, “Dynamic maximum block size by miner vote.” `https://github.com/jgarzik/bip100/blob/master/bip-0100.mediawiki`, 2015. Online; Accessed: May 24, 2021.

[51] G. Andresen, “Increase maximum block size.” `https://github.com/bitcoin/bips/blob/master/bip-0101.mediawiki`, 2015. Online; Accessed: May 24, 2021.

[52] J. Garzik, “Block size increase to 2MB.” `https://github.com/bitcoin/bips/blob/master/bip-0102.mediawiki`, 2015. Online; Accessed: May 24, 2021.

[53] P. Wuille, “Block size following technological growth.” `https://github.com/bitcoin/bips/blob/master/bip-0103.mediawiki`, 2015. Online; Accessed: May 24, 2021.

[54] M. Corallo, “Compact Block Relay.” `https://github.com/bitcoin/bips/blob/master/bip-0152.mediawiki`, 2016. Online; Accessed: May 24, 2021.

[55] A. P. Ozisik, G. Andresen, B. N. Levine, D. Tapp, G. Bissias, and S. Katkuri, "Graphene: Efficient Interactive Set Reconciliation Applied to Blockchain Propagation," in *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, (New York, NY, USA), p. 303–317, Association for Computing Machinery, 2019.

[56] P. Tschipper, "BUIP010 (passed): Xtreme Thinblocks." `https://bitco.in/forum/threads/buip010-passed-xtreme-thinblocks.774/`, 2016. Online; Accessed: Nov 11, 2021.

[57] "Detailed Protocol Design for Xtreme Thin blocks (Xthinblocks)." `https://github.com/BitcoinUnlimited/BitcoinUnlimited/blob/release/doc/bu-xthin-protocol.md`, 2016. Online; Accessed: Nov 11, 2021.

[58] M. Dotan, Y.-A. Pignolet, S. Schmid, S. Tochner, and A. Zohar, "SOK: Cryptocurrency Networking Context, State-of-the-Art, Challenges," in *Proceedings of the 15th International Conference on Availability, Reliability and Security*, ARES '20, (New York, NY, USA), Association for Computing Machinery, 2020.

[59] A. E. Gencer, S. Basu, I. Eyal, R. van Renesse, and E. G. Sirer, "Decentralization in Bitcoin and Ethereum Networks," in *Financial Cryptography and Data Security* (S. Meiklejohn and K. Sako, eds.), (Berlin, Heidelberg), pp. 439–457, Springer Berlin Heidelberg, 2018.

[60] U. Klarman, S. Basu, A. Kuzmanovic, and E. G. Sirer, "bloxroute: A scalable trustless blockchain distribution network whitepaper." `https://bloxroute.com/wp-content/uploads/2019/11/bloXrouteWhitepaper.pdf`, 2018. Online; Accessed: Nov 10, 2021.

[61] N. Chawla, H. W. Behrens, D. Tapp, D. Boscovic, and K. S. Candan, "Velocity: Scalability Improvements in Block Propagation Through Rateless Erasure Coding," in *2019 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, pp. 447–454, 2019.

[62] L. Zhang, T. Wang, and S. C. Liew, "Speeding up Block Propagation in Blockchain Network: Uncoded and Coded Designs," *CoRR*, vol. abs/2101.00378, 2021.

[63] G. Andresen, "[bitcoin-dev] Weak block thoughts...." `https://lists.linuxfoundation.org/pipermail/bitcoin-dev/2015-Sep/011157.html`, 2015. Online; Accessed: Nov 11, 2021.

[64] J. Huang, L. Tan, S. Mao, and K. Yu, "Blockchain Network Propagation Mechanism Based on P4P Architecture," *Security and Communication Networks*, vol. 2021, p. 8363131, Aug 2021.

[65] M. Jin, X. Chen, and S.-J. Lin, “Reducing the Bandwidth of Block Propagation in Bitcoin Network With Erasure Coding,” *IEEE Access*, vol. 7, pp. 175606–175613, 2019.

[66] Z. Lihao, T. Wang, and S. C. Liew, “Speeding up Block Propagation in Blockchain Network: Uncoded and Coded Designs.” `https://www.researchgate.net/publication/348212522_Speeding_up_Block_Propagation_in_Blockchain_Network_Uncoded_and_Coded_Designs`, 01 2021. Online; Accessed: Nov 11, 2021.

[67] E. Rohrer and F. Tschorsch, “Kadcast: A Structured Approach to Broadcast in Blockchain Networks,” in *Proceedings of the 1st ACM Conference on Advances in Financial Technologies*, AFT ’19, (New York, NY, USA), p. 199–213, Association for Computing Machinery, 2019.

[68] 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, 2019.

[69] M. A. Imtiaz, D. Starobinski, A. Trachtenberg, and N. Younis, “Churn in the Bitcoin Network,” *IEEE Transactions on Network and Service Management*, vol. 18, no. 2, pp. 1598–1615, 2021.

[70] L. Kiffer, A. Salman, D. Levin, A. Mislove, and C. Nita-Rotaru, “Under the Hood of the Ethereum Gossip Protocol,” in *Proceedings of the 2021 International Conference on Financial Cryptography and Data Security (FC’21)*, 2021.

[71] D. Mechkaroska, V. Dimitrova, and A. Popovska-Mitrovikj, “Analysis of the Possibilities for Improvement of BlockChain Technology,” in *2018 26th Telecommunications Forum (TELFOR)*, pp. 1–4, 2018.

[72] M. Essaid, H. W. Kim, W. Guil Park, K. Y. Lee, S. Jin Park, and H. T. Ju, “Network Usage of Bitcoin Full Node,” in *2018 International Conference on Information and Communication Technology Convergence (ICTC)*, pp. 1286–1291, 2018.

[73] D. Perard, J. Lacan, Y. Bachy, and J. Detchart, “Erasure Code-Based Low Storage Blockchain Node,” in *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, pp. 1622–1627, 2018.

[74] T. Dryja, “Utreexo: A dynamic hash-based accumulator optimized for the Bitcoin UTXO set.” Cryptology ePrint Archive, Report 2019/611, 2019. `https://ia.cr/2019/611`.

[75] M. Florian, S. Henningsen, S. Beaucamp, and B. Scheuermann, “Erasing Data from Blockchain Nodes,” in *2019 IEEE European Symposium on Security and Privacy Workshops (EuroS PW)*, pp. 367–376, 2019.

[76] F. Tschorsch and B. Scheuermann, “Bitcoin and Beyond: A Technical Survey on Decentralized Digital Currencies,” *IEEE Communications Surveys Tutorials*, vol. 18, no. 3, pp. 2084–2123, 2016.

[77] E. developers, “NODES AND CLIENTS.” https://ethereum.org/en/developers/docs/nodes-and-clients/#network-benefits. Online; Accessed: Aug 18, 2021.

[78] Y. Wang, “A Blockchain System with Lightweight Full Node Based on Dew Computing,” *Internet of Things*, vol. 11, p. 100184, 2020.

[79] O. I. Oluwasuji, O. Malik, J. Zhang, and S. D. Ramchurn, “Solving the fair electric load shedding problem in developing countries,” *Autonomous Agents and Multi-Agent Systems*, vol. 34, p. 12, Dec 2019.

[80] A. Hafeez, W. A. Khan, and M. A. Rahman, “Development of Financial Model to Solarize Public Institutes in Pakistan,” in *2021 4th International Conference on Energy Conservation and Efficiency (ICECE)*, pp. 1–10, 2021.

[81] T. H. Meles, “Impact of power outages on households in developing countries: Evidence from Ethiopia,” *Energy Economics*, vol. 91, p. 104882, 2020.

[82] H. H. Alhelou, M. E. Hamedani-Golshan, T. C. Njenda, and P. Siano, “A Survey on Power System Blackout and Cascading Events: Research Motivations and Challenges,” *Energies*, vol. 12, pp. 1–28, February 2019.

[83] G. R. Timilsina, P. Sapkota, and J. Steinbuks, “How much has Nepal lost in the last decade due to load shedding? An economic assessment using a CGE model,” *Policy Research Working Papers, World Bank, Washington, DC*, Jun 2018.

[84] IEA, IRENA, UNSD, WB, and WHO, “Tracking SDG7: The Energy Progress Report.” https://irena.org/publications/2019/May/Tracking-SDG7-The-Energy-Progress-Report-2019, 2019. Online; Accessed: Aug 14, 2021.

[85] L. Odarno, “1.2 Billion People Lack Electricity. Increasing Supply Alone Won’t Fix the Problem.” https://www.wri.org/insights/12-billion-people-lack-electricity-increasing-supply-alone-wont-fix-problem, Mar 2017. Online; Accessed: Aug 14, 2021.

[86] R. Fetter, A. Fuller, J. Porcaro, and C. Sinai, "You can't fight pandemics without power—electric power." `https://www.brookings.edu/blog/future-development/2020/06/05/you-cant-fight-pandemics-without-power-electric-power/`, Jun 2020. Online; Accessed: Aug 14, 2021.

[87] P. Cramton, "Lessons from the 2021 Texas electricity crisis," *Utility Dive*, May 2021.

[88] M. Rae, *The Texas Energy Crisis: Has Deregulation Hurt Consumers?* London: SAGE Publications: SAGE Business Cases Originals, 2021.

[89] J. Bialek, "What does the GB power outage on 9 August 2019 tell us about the current state of decarbonised power systems?," *Energy Policy*, vol. 146, p. 111821, 2020.

[90] G. J. Rubin and M. B. Rogers, "Behavioural and psychological responses of the public during a major power outage: A literature review," *International Journal of Disaster Risk Reduction*, vol. 38, p. 101226, 2019.

[91] W. Li, J. Zhou, and X. Hu, "Comparison of transmission equipment outage performance in Canada, USA and China," in *2008 IEEE Canada Electric Power Conference*, pp. 1–8, 2008.

[92] A. Lindstrom and S. Hoff, "U.S. customers experienced an average of nearly six hours of power interruptions in 2018." `https://www.eia.gov/todayinenergy/detail.php?id=43915`, Jun 2020. Online; Accessed: Aug 14, 2021.

[93] A. Muir and J. Lopatto, "Final report on the Aug 14, 2003 blackout in the United States and Canada : causes and recommendations." `https://www.energy.gov/sites/default/files/oeprod/DocumentsandMedia/BlackoutFinal-Web.pdf`, Apr 2004. Online; Accessed: Nov 27, 2021.

[94] R. Perez, M. Kmiecik, T. Hoff, J. Williams, C. Herig, S. Letendre, and R. Margolis, "Availability of dispersed Photovoltaic resource during the August 14th 2003 northeast power outage," *Proceedings of the American Solar Energy Society, Portland, OR*, 2004.

[95] S. Adderly, "Reviewing power outage trends, electric reliability indices and smart grid funding," Master's thesis, The University of Vermont and State Agricultural College, 2016.

[96] GENERAC, "Current power outages - USA." `https://www.generac.com/poweroutagecentral`, 2021. Online; Accessed: Aug 14, 2021.

[97] GENERAC, “Current power outages - Canada.” https://www.generac.com/be-prepared/power-outages/power-outage-tracker-canada, 2021. Online; Accessed: Aug 14, 2021.

[98] R. Wood, “Power returns after major outage in eastern Sydney.” https://www.9news.com.au/national/sydney-power-outage-ausgrid-website-crashes/82674013-2a60-438a-a589-7bf89a0acc67, Jul 2021. Online; Accessed: Aug 14, 2021.

[99] D. Stutzbach and R. Rejaie, “Understanding churn in peer-to-peer networks,” in *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, pp. 189–202, ACM, 2006.

[100] D. Stutzbach and R. Rejaie, “Towards a better understanding of churn in peer-to-peer networks,” Tech. Rep. CIS-TR-04-06, Department of Computer Science, University of Oregon, http://ix.cs.uoregon.edu/~reza/PUB/tr04-06.pdf, 2004.

[101] O. Herrera and T. Znati, “Modeling Churn in P2P Networks,” in *40th Annual Simulation Symposium (ANSS’07)*, pp. 33–40, 2007.

[102] Z. Yao, D. Leonard, X. Wang, and D. Loguinov, “Modeling Heterogeneous User Churn and Local Resilience of Unstructured P2P Networks,” in *Proceedings of the 2006 IEEE International Conference on Network Protocols*, pp. 32–41, 2006.

[103] X. Hei, C. Liang, J. Liang, Y. Liu, and K. W. Ross, “A measurement study of a large-scale P2P IPTV system,” *IEEE transactions on multimedia*, vol. 9, no. 8, pp. 1672–1687, 2007.

[104] D. Yang, Y.-x. Zhang, H.-k. Zhang, T.-Y. Wu, and H.-C. Chao, “Multi-factors oriented study of P2P Churn,” *International Journal of Communication Systems*, vol. 22, no. 9, pp. 1089–1103, 2009.

[105] F. Lin, C. Chen, and H. Zhang, “Characterizing Churn in Gnutella Network in a New Aspect,” in *2008 9th International Conference for Young Computer Scientists*, (Los Alamitos, CA, USA), pp. 305–309, IEEE Computer Society, Nov 2008.

[106] I. Eyal, A. E. Gencer, E. G. Sirer, and R. V. Renesse, “Bitcoin-NG: A Scalable Blockchain Protocol,” in *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, (Santa Clara, CA), pp. 45–59, USENIX Association, Mar. 2016.

[107] M. Shayan, C. Fung, C. J. M. Yoon, and I. Beschastnikh, "Biscotti: A Blockchain System for Private and Secure Federated Learning," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 7, pp. 1513–1525, 2021.

[108] S. Pešić, M. Radovanović, M. Ivanović, M. Tošić, O. Iković, and D. Bošković, "Hyperledger Fabric Blockchain as a Service for the IoT: Proof of Concept," in *Model and Data Engineering* (K.-D. Schewe and N. K. Singh, eds.), (Cham), pp. 172–183, Springer International Publishing, 2019.

[109] Y. Hu, A. Manzoor, P. Ekparinya, M. Liyanage, K. Thilakarathna, G. Jourjon, and A. Seneviratne, "A Delay-Tolerant Payment Scheme Based on the Ethereum Blockchain," *IEEE Access*, vol. 7, pp. 33159–33172, 2019.

[110] T. Neudecker, P. Andelfinger, and H. Hartenstein, "A simulation model for analysis of attacks on the Bitcoin peer-to-peer network," in *2015 IFIP/IEEE International Symposium on Integrated Network Management (IM)*, pp. 1327–1332, 2015.

[111] M. Apostolaki, A. Zohar, and L. Vanbever, "Hijacking Bitcoin: Routing Attacks on Cryptocurrencies," in *2017 IEEE Symposium on Security and Privacy (SP)*, pp. 375–392, 2017.

[112] G. O. Karame, E. Androulaki, and S. Capkun, "Double-spending fast payments in bitcoin," in *Proceedings of the 2012 ACM conference on Computer and communications security*, pp. 906–917, ACM, 2012.

[113] T. Neudecker, "Characterization of the Bitcoin Peer-to-Peer Network (2015-2018)." `http://dsn.tm.kit.edu/bitcoin/publications/bitcoin_network_characterization.pdf`, 2019.

[114] S. K. Kim, Z. Ma, S. Murali, J. Mason, A. Miller, and M. Bailey, "Measuring Ethereum Network Peers," in *Proceedings of the Internet Measurement Conference 2018*, IMC '18, (New York, NY, USA), p. 91–104, Association for Computing Machinery, 2018.

[115] L. Zhang, B. Lee, Y. Ye, and Y. Qiao, "Evaluation of Ethereum End-to-end Transaction Latency," in *2021 11th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*, pp. 1–5, 2021.

[116] Y. Shahsavari, K. Zhang, and C. Talhi, "A Theoretical Model for Block Propagation Analysis in Bitcoin Network," *IEEE Transactions on Engineering Management*, pp. 1–18, 2020.

[117] S. Maeng, M. Essaid, C. Lee, S. Park, and H. Ju, “Visualization of Ethereum P2P network topology and peer properties,” *International Journal of Network Management*, vol. 31, no. 6, p. e2175, 2021.

[118] J. A. Donet Donet, C. Pérez-Solà, and J. Herrera-Joancomartí, “The bitcoin p2p network,” in *Financial Cryptography and Data Security* (R. Böhme, M. Brenner, T. Moore, and M. Smith, eds.), (Berlin, Heidelberg), pp. 87–102, Springer Berlin Heidelberg, 2014.

[119] K. Dae-Yong, E. Meryam, and J. Hongtaek, “Examining Bitcoin mempools Resemblance Using Jaccard Similarity Index,” in *2020 21st Asia-Pacific Network Operations and Management Symposium (APNOMS)*, pp. 287–290, 2020.

[120] J. Mišić, V. B. Mišić, and X. Chang, “Performance of Bitcoin Network With Synchronizing Nodes and a Mix of Regular and Compact Blocks,” *IEEE Transactions on Network Science and Engineering*, vol. 7, no. 4, pp. 3135–3147, 2020.

[121] M. A. Imtiaz, “bitcoin-releases.” `https://github.com/nislab/bitcoin-releases/tree/tnsm-churn`, 2020.

[122] M. A. Imtiaz, “bitcoin-logs.” `https://github.com/nislab/bitcoin-logs/tree/tnsm-churn`, 2020.

[123] A. Pinar Ozisik, G. Andresen, G. Bissias, A. Houmansadr, and B. Levine, “Graphene: A New Protocol for Block Propagation Using Set Reconciliation,” in *Data Privacy Management, Cryptocurrencies and Blockchain Technology* (J. Garcia-Alfaro, G. Navarro-Arribas, H. Hartenstein, and J. Herrera-Joancomartí, eds.), (Cham), pp. 420–428, Springer International Publishing, 2017.

[124] M. A. Imtiaz, “bitcoin-releases.” `https://github.com/nislab/bitcoin-releases/tree/wip`, 2021.

[125] M. A. Imtiaz, “bitcoin-logs.” `https://github.com/nislab/bitcoin-logs/tree/wip`, 2021.

[126] Bitcoin Wiki, “Block.” `https://en.bitcoin.it/wiki/Block`, 2016. Online; Accessed: Nov 17, 2021.

[127] “Block hashing algorithm.” `https://en.bitcoin.it/wiki/Block_hashing_algorithm`. Online; Accessed: Nov 17, 2021.

[128] E. developers, "BLOCKS." https://ethereum.org/en/developers/docs/blocks/. Online; Accessed: Nov 17, 2021.

[129] "Order of transactions within a block." https://bitcoin.stackexchange.com/q/23035, Mar 2014. Online; Accessed: Aug 9, 2020.

[130] R. C. Merkle, "A Digital Signature Based on a Conventional Encryption Function," in *Advances in Cryptology — CRYPTO '87* (C. Pomerance, ed.), (Berlin, Heidelberg), pp. 369–378, Springer Berlin Heidelberg, 1988.

[131] V. Buterin, "Merkling in Ethereum." https://blog.ethereum.org/2015/11/15/merkling-in-ethereum/, Nov 2015. Online; Accessed: Nov 18, 2021.

[132] S. Gorbunov, L. Reyzin, H. Wee, and Z. Zhang, "Pointproofs: Aggregating Proofs for Multiple Vector Commitments," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20, (New York, NY, USA), p. 2007–2023, Association for Computing Machinery, 2020.

[133] C. Cachin and M. Vukolic, "Blockchain Consensus Protocols in the Wild (Keynote Talk)," in *31st International Symposium on Distributed Computing (DISC 2017)* (A. W. Richa, ed.), vol. 91 of *Leibniz International Proceedings in Informatics (LIPIcs)*, (Dagstuhl, Germany), pp. 1:1–1:16, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017.

[134] S. Zhang and J.-H. Lee, "Analysis of the main consensus protocols of blockchain," *Information and Communications Technology (ICT) Express*, vol. 6, no. 2, pp. 93–97, 2020.

[135] Y. Xiao, N. Zhang, W. Lou, and Y. T. Hou, "A Survey of Distributed Consensus Protocols for Blockchain Networks," *IEEE Communications Surveys Tutorials*, vol. 22, no. 2, pp. 1432–1465, 2020.

[136] QuantumMechanic, "Proof of stake instead of proof of work." https://bitcointalk.org/index.php?topic=27787.0. Online; Accessed: Aug 10, 2021.

[137] I. Bentov, A. Gabizon, and A. Mizrahi, "Cryptocurrencies Without Proof of Work," in *Financial Cryptography and Data Security* (J. Clark, S. Meiklejohn, P. Y. Ryan, D. Wallach, M. Brenner, and K. Rohloff, eds.), (Berlin, Heidelberg), pp. 142–157, Springer Berlin Heidelberg, 2016.

[138] Hyperledger, “Proof of Elapsed Time (PoET).” https://sawtooth.hyperledger.org/docs/core/nightly/0-8/introduction.html#proof-of-elapsed-time-poet. Online; Accessed: Aug 10, 2021.

[139] “Quorum Whitepaper.” https://github.com/ConsenSys/quorum/blob/master/docs/Quorum%20Whitepaper%20v0.2.pdf. Online; Accessed: Nov 24, 2021.

[140] O. Beigel, “The Complete Guide to Bitcoin Fees.” https://99bitcoins.com/bitcoin/fees/. Online; Accessed: Dec 8, 2019.

[141] “Confirmed Transactions Per Day.” https://www.blockchain.com/en/charts/n-transactions?timespan=180days. Online; Accessed: Nov 24, 2021.

[142] “Total Number of Transactions.” https://www.blockchain.com/charts/n-transactions-total?timespan=all. Online; Accessed: Nov 24, 2021.

[143] “Network Difficulty.” https://www.blockchain.com/charts/difficulty. Online; Accessed: Nov 24, 2021.

[144] “Total Hash Rate (TH/s).” https://www.blockchain.com/charts/hash-rate. Online; Accessed: Nov 24, 2021.

[145] A. Stone, “`bu0.12.0`.” https://github.com/BitcoinUnlimited/BitcoinUnlimited/releases/tag/bu0.12.0, 2016. Online; Accessed: Apr 7, 2021.

[146] C. Decker, *On the Scalability and Security of Bitcoin*. PhD thesis, ETH Zurich, Zurich, https://doi.org/10.3929/ethz-a-010619000, 2016.

[147] B. Magazine, “What is the block size limit?.” https://bitcoinmagazine.com/guides/what-is-the-bitcoin-block-size-limit, 2020. Online; Accessed: Apr 7, 2021.

[148] Blockchain.com, “Confirmed transactions per day.” https://www.blockchain.com/charts/n-transactions. Online; Accessed: Apr 7, 2021.

[149] B. Unlimited, “Bitcoin Unlimited FAQ.” https://www.bitcoinunlimited.info/faq/what-is-bu. Online; Accessed: Apr 7, 2021.

[150] Blockchain.com, “Average transactions per block.” https://www.blockchain.com/charts/n-transactions-per-block. Online; Accessed: Apr 7, 2021.

[151] "Block #681,765."
https://explorer.bitcoinunlimited.info/block-height/681765.
Online; Accessed: Nov 24, 2021.

[152] B. C. Explorer, "Block stats."
https://explorer.bitcoinunlimited.info/block-stats. Online; Accessed: Apr 7, 2021.

[153] B. H. Bloom, "Space/Time Trade-Offs in Hash Coding with Allowable Errors," *Communications of the ACM*, vol. 13, p. 422–426, July 1970.

[154] Li Fan, Pei Cao, J. Almeida, and A. Z. Broder, "Summary cache: a scalable wide-area Web cache sharing protocol," *IEEE/ACM Transactions on Networking*, vol. 8, no. 3, pp. 281–293, 2000.

[155] S. Agarwal and A. Trachtenberg, "Approximating the number of differences between remote sets," in *2006 IEEE Information Theory Workshop - ITW '06 Punta del Este*, pp. 217–221, 2006.

[156] M. T. Goodrich and M. Mitzenmacher, "Invertible bloom lookup tables," in *2011 49th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, pp. 792–799, 2011.

[157] D. Eppstein and M. T. Goodrich, "Straggler Identification in Round-Trip Data Streams via Newton's Identities and Invertible Bloom Filters," *IEEE Transactions on Knowledge and Data Engineering*, vol. 23, no. 2, pp. 297–306, 2011.

[158] D. Eppstein, M. T. Goodrich, F. Uyeda, and G. Varghese, "What's the Difference? Efficient Set Reconciliation without Prior Context," in *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, (New York, NY, USA), p. 218–229, Association for Computing Machinery, 2011.

[159] A. Suisani, "`Bitcoin Unlimited - Bitcoin Cash release 1.6.0.0`."
https://github.com/BitcoinUnlimited/BitcoinUnlimited/releases/tag/bucash1.6.0.0, 2019. Online; Accessed: Apr 9, 2021.

[160] "`net_processing.cpp`." https://gitlab.com/bitcoinunlimited/BCHUnlimited/-/blob/BCHunlimited1.9.0.1/src/net_processing.cpp#L399-408.
Online; Accessed: Apr 9, 2021.

[161] G. Bissias, "`graphene-specification-v2.2.mediawiki`."
https://gitlab.com/bitcoinunlimited/BCHUnlimited/-/blob/dev/doc/graphene-specification-v2.2.mediawiki, 2019. Online; Accessed: Apr 10, 2021.

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

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

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

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

[166] H. Kalodner, M. Möser, K. Lee, S. Goldfeder, M. Plattner, A. Chator, and A. Narayanan, "BlockSci: Design and applications of a blockchain analysis platform," in *29th USENIX Security Symposium (USENIX Security 20)*, pp. 2721–2738, USENIX Association, Aug. 2020.

[167] J. Augustine, G. Pandurangan, and P. Robinson, "Distributed algorithmic foundations of dynamic networks," *ACM SIGACT News*, vol. 47, no. 1, pp. 69–98, 2016.

[168] T. Jacobs and G. Pandurangan, "Stochastic analysis of a churn-tolerant structured peer-to-peer scheme," *Peer-to-Peer Networking and Applications*, vol. 6, no. 1, pp. 1–14, 2013.

[169] "How does a node get information from other nodes?." `https://bitcoin.stackexchange.com/questions/70621/how-does-a-node-get-information-from-other-nodes/70623`. Online; Accessed: Oct 17, 2019.

[170] J. Mišić, V. B. Mišić, and X. Chang, "On the benefits of compact blocks in Bitcoin," in *ICC 2020-2020 IEEE International Conference on Communications (ICC)*, pp. 1–6, IEEE, 2020.

[171] S. G. Motlagh, J. Misic, and V. B. Misic, "Modeling of Churn Process in Bitcoin Network," in *2020 International Conference on Computing, Networking and Communications (ICNC)*, pp. 686–691, IEEE, 2020.

[172] S. G. Motlagh, J. Mišić, and V. B. Mišić, "Impact of Node Churn in the Bitcoin Network," *IEEE Transactions on Network Science and Engineering*, vol. 7, no. 3, pp. 2104–2113, 2020.

[173] S. G. Motlagh, J. Mišić, and V. B. Mišić, "Impact of Node Churn in the Bitcoin Network with Compact Blocks," in *GLOBECOM 2020 - 2020 IEEE Global Communications Conference*, pp. 1–6, 2020.

[174] S. G. Motlagh, J. Mišić, and V. B. Mišić, "An analytical model for churn process in Bitcoin network with ordinary and relay nodes," *Peer-to-Peer Networking and Applications*, vol. 13, pp. 1931–1942, 2020.

[175] M. Saad, S. Chen, and D. Mohaisen, "Root Cause Analyses for the Deteriorating Bitcoin Network Synchronization," in *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*, pp. 239–249, 2021.

[176] G. Naumenko, G. Maxwell, P. Wuille, A. Fedorova, and I. Beschastnikh, "Erlay: Efficient Transaction Relay for Bitcoin," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pp. 817–831, 2019.

[177] "Bitcoin Orphan Transactions and CVE-2012-3789." `https://cryptoservices.github.io/fde/2018/12/14/bitcoin-orphan-TX-CVE.html`. Online; Accessed: Feb 12, 2020.

[178] "DoS fix for mapOrphanTransactions." `https://github.com/bitcoin/bitcoin/pull/911`. Online; Accessed: Feb 12, 2020.

[179] A. Yeow, "Bitnodes." `https://bitnodes.earn.com`. Online; Accessed: Nov 15, 2018.

[180] A. Yeow, "Bitnodes API v1.0." `https://bitnodes.earn.com/api/`. Online; Accessed: Dec 17, 2018.

[181] "Protocol Documentation." `https://en.bitcoin.it/wiki/Protocol_documentation`, 2018. Online; Accessed: May 17, 2018.

[182] "Full node, NODE_NETWORK_LIMITED (1037) what does it mean?." `https://www.reddit.com/r/Bitcoin/comments/8wkuod/full_node_node_network_limited_1037_what_does_it/`. Online; Accessed: Sep 19, 2020.

[183] J. Falkner, M. Piatek, J. P. John, A. Krishnamurthy, and T. Anderson, "Profiling a million user DHT," in *Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*, pp. 129–134, ACM, 2007.

[184] F. E. Bustamante and Y. Qiao, "Friendships that last: Peer lifespan and its role in P2P protocols," in *Web content caching and distribution*, pp. 233–246, Springer, 2004.

[185] K. P. Gummadi, R. J. Dunn, S. Saroiu, S. D. Gribble, H. M. Levy, and J. Zahorjan, “Measurement, modeling, and analysis of a peer-to-peer file-sharing workload,” *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5, pp. 314–329, 2003.

[186] J. Li, J. Stribling, R. Morris, and M. F. Kaashoek, “Bandwidth-efficient management of DHT routing tables,” in *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*, pp. 99–114, USENIX Association, 2005.

[187] S. Sen and J. Wang, “Analyzing peer-to-peer traffic across large networks,” in *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurment*, pp. 137–150, ACM, 2002.

[188] S. Foss, D. Korshunov, and S. Zachary, “Heavy-Tailed and Long-Tailed Distributions,” in *An Introduction to Heavy-Tailed and Subexponential Distributions*, pp. 7–42, New York, NY: Springer New York, 2013.

[189] J. P. Nolan, “Maximum likelihood estimation and diagnostics for stable distributions,” in *Lévy processes*, pp. 379–400, Springer, 2001.

[190] T. M. Inc., “Fit probability distribution object to data - MATLAB fitdist.” `https://www.mathworks.com/help/stats/fitdist.html`. Online; Accessed: Nov 06, 2018.

[191] “Evaluating Goodness of Fit.” `https://www.mathworks.com/help/curvefit/evaluating-goodness-of-fit.html#bq_5kwr-7`. Online; Accessed: Nov 18, 2018.

[192] “Coefficient of Determination (R-squared) Explained.” `https://towardsdatascience.com/coefficient-of-determination-r-squared-explained-db32700d924e`. Online; Accessed: Dec 2, 2018.

[193] “RMS Error.” `http://statweb.stanford.edu/~susan/courses/s60/split/node60.html`. Online; Accessed: Dec 2, 2018.

[194] S. Ross, *Stochastic Processes*, p. 114. Wiley series in probability and mathematical statistics, Wiley, second ed., 1995.

[195] M. G. Reed, P. F. Syverson, and D. M. Goldschlag, “Anonymous connections and onion routing,” *IEEE Journal on Selected areas in Communications*, vol. 16, no. 4, pp. 482–494, 1998.

[196] J. Benesty, J. Chen, Y. Huang, and I. Cohen, “Pearson correlation coefficient,” in *Noise reduction in speech processing*, pp. 1–4, Springer, 2009.

[197] "Pearson Correlations." https://www.spss-tutorials.com/pearson-correlation-coefficient/. Online; Accessed: Sep 23, 2020.

[198] "Bitcoin Developer Reference." https://bitcoin.org/en/developer-reference#remote-procedure-calls-rpcs, 2017. Online; Accessed: Nov 17, 2018.

[199] "Bitcoin Core :: setban (0.16.0 RPC)." https://bitcoincore.org/en/doc/0.16.0/rpc/network/setban/. Online; Accessed: Nov 17, 2018.

[200] Y. Minsky, A. Trachtenberg, and R. Zippel, "Set reconciliation with nearly optimal communication complexity," *IEEE Transactions on Information Theory*, vol. 49, no. 9, pp. 2213–2218, 2003.

[201] "`miner.cpp`." https://github.com/bitcoin/bitcoin/blob/master/src/miner.cpp#L321. Online; Accessed: May 11, 2020.

[202] "`txmempool.h`." https://github.com/bitcoin/bitcoin/blob/master/src/txmempool.h, 2018.

[203] "`std::map`." https://en.cppreference.com/w/cpp/container/map. Online; Accessed: Sep 23, 2020.

[204] "`net.h`." https://github.com/bitcoin/bitcoin/blob/master/src/net.h#L50. Online; Accessed: May 12, 2020.

[205] "btcd." https://github.com/btcsuite/btcd. Online; Accessed: Sep 23, 2020.

[206] "Bitcoin Knots." https://bitcoinknots.org/. Online; Accessed: Sep 23, 2020.

[207] "Bitcoin Knots (source)." https://github.com/bitcoinknots/bitcoin/tree/v0.20.1.knots20200815. Online; Accessed: Sep 23, 2020.

[208] "Libbitcoin Node." https://github.com/libbitcoin/libbitcoin-node. Online; Accessed: Sep 23, 2020.

[209] "bitcoinj." https://bitcoinj.org/. Online; Accessed: Sep 23, 2020.

[210] "5 Bitcoin Core Alternatives That Don't Fork the Blockchain." https://bitcoin.eu/bitcoin-core-alternatives-dont-fork-blockchain/. Online; Accessed: Sep 23, 2020.

[211] an4s, "Unreachable code when checking for valid Merkle root in compact block processing." https://gitlab.com/bitcoinunlimited/BCHUnlimited/-/issues/2226. Online; Accessed: May 21, 2021.

[212] "Transaction Size." https://bitcoinvisuals.com/chain-tx-size. Online; Accessed: Oct 25, 2021.

[213] "Mempool Summary." https://explorer.bitcoinunlimited.info/mempool-summary. Online; Accessed: Oct 25, 2021.

[214] C. Spearman, "The Proof and Measurement of Association between Two Things," *The American Journal of Psychology*, vol. 100, no. 3/4, pp. 441–471, 1987.

[215] D. Zwillinger, *CRC standard probability and statistics tables and formulae*. Boca Raton: Chapman & Hall/CRC, 2000.

[216] "scipy.stats.spearmanr." https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.spearmanr.html. Online; Accessed: Oct 11, 2021.

[217] F. Wilcoxon, "Individual Comparisons by Ranking Methods," *Biometrics Bulletin*, vol. 1, no. 6, pp. 80–83, 1945.

[218] W. W. LaMorte, "PH717 Module 9 - Correlation and Regression: Evaluating Association Between Two Continuous Variables." https://sphweb.bumc.bu.edu/otlt/MPH-Modules/PH717-QuantCore/PH717-Module9-Correlation-Regression/PH717-Module9-Correlation-Regression4.html. Online; Accessed: May 14, 2021.

[219] D. Ron and A. Shamir, "Quantitative Analysis of the Full Bitcoin Transaction Graph," in *Financial Cryptography and Data Security* (A.-R. Sadeghi, ed.), (Berlin, Heidelberg), pp. 6–24, Springer Berlin Heidelberg, 2013.

[220] 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.

[221] M. Fleder, M. S. Kester, and S. Pillai, "Bitcoin Transaction Graph Analysis," *CoRR*, vol. abs/1502.01657, 2015.

[222] 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, 2015.

[223] 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, 2013.

[224] A. Greaves and B. Au, "Using the Bitcoin transaction graph to predict the price of Bitcoin." http://snap.stanford.edu/class/cs224w-2015/projects_2015/Using_the_Bitcoin_Transaction_Graph_to_Predict_the_Price_of_Bitcoin.pdf, 2015. Online; Accessed: Nov 27, 2021.

[225] 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.

[226] E. Androulaki, G. O. Karame, M. Roeschlin, T. Scherer, and S. Capkun, "Evaluating User Privacy in Bitcoin," in *Financial Cryptography and Data Security* (A.-R. Sadeghi, ed.), (Berlin, Heidelberg), pp. 34–51, Springer Berlin Heidelberg, 2013.

[227] T. Ruffing and P. Moreno-Sanchez, "ValueShuffle: Mixing Confidential Transactions for Comprehensive Transaction Privacy in Bitcoin," in *Financial Cryptography and Data Security* (M. Brenner, K. Rohloff, J. Bonneau, A. Miller, P. Y. Ryan, V. Teague, A. Bracciali, M. Sala, F. Pintore, and M. Jakobsson, eds.), (Cham), pp. 133–154, Springer International Publishing, 2017.

[228] J. Herrera-Joancomartí and C. Pérez-Solà, "Privacy in Bitcoin Transactions: New Challenges from Blockchain Scalability Solutions," in *Modeling Decisions for Artificial Intelligence* (V. Torra, Y. Narukawa, G. Navarro-Arribas, and C. Yañez, eds.), (Cham), pp. 26–44, Springer International Publishing, 2016.

[229] Q. Wang, B. Qin, J. Hu, and F. Xiao, "Preserving transaction privacy in bitcoin," *Future Generation Computer Systems*, vol. 107, pp. 793–804, 2020.

[230] 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.

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

[232] Y. Kawase and S. Kasahara, "Transaction-Confirmation Time for Bitcoin: A Queueing Analytical Approach to Blockchain Mechanism," in *Queueing Theory and Network Applications* (W. Yue, Q.-L. Li, S. Jin, and Z. Ma, eds.), (Cham), pp. 75–88, Springer International Publishing, 2017.

[233] Y. Sompolinsky and A. Zohar, "Accelerating Bitcoin's Transaction Processing. Fast Money Grows on Trees, Not Chains.," *IACR Cryptology ePrint Archive*, vol. 2013, p. 881, 2013.

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

[235] 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, 2016.

[236] "BEST PAPER, RUNNER UP AND TNSM INVITATIONS." `https://icbc2020.ieee-icbc.org/best-paper-runner-and-tnsm-inviations`, 2020. Online; Accessed: Nov 27, 2021.

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

[238] "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: Dec 3, 2019.

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

[240] 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.

[241] M. A. Imtiaz, D. Starobinski, A. Trachtenberg, and N. Younis, "Churn in the Bitcoin Network," *IEEE Transactions on Network and Service Management*, vol. 18, no. 2, pp. 1598–1615, 2021.

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

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

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

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

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

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

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

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

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

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

[252] J. Poon and T. Dryja, "The Bitcoin Lightning Network: Scalable Off-Chain Instant Payments." https://lightning.network/lightning-network-paper.pdf, Jan 2016. Online; Accessed: Nov 22, 2021.

[253] V. Buterin, "Sharding-FAQs." https://eth.wiki/sharding/Sharding-FAQs. Online; Accessed: Nov 22, 2021.

[254] J. Poon and V. Buterin, "Plasma: Scalable Autonomous Smart Contracts." https://plasma.io/plasma.pdf, Aug 2017. Online; Accessed: Nov 22, 2021.

[255] G. Malavolta, P. Moreno-Sanchez, A. Kate, M. Maffei, and S. Ravi, "Concurrency and privacy with payment-channel networks," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, (New York, NY, USA), p. 455–471, Association for Computing Machinery, 2017.

[256] L. Gudgeon, P. Moreno-Sanchez, S. Roos, P. McCorry, and A. Gervais, "SoK: Layer-Two Blockchain Protocols," in *Financial Cryptography and Data Security*, (Cham), pp. 201–226, Springer International Publishing, 2020.

Curriculum Vitae

MUHAMMAD ANAS IMTIAZ

anasimtiaz.pk 🔗 | in maimtiaz
me@anasimtiaz.pk ✉ | 📱 +1 (617) 371-7268

EDUCATION

M.S./Ph.D. Computer Engineering *(expected)*
Boston University, Boston MA, USA January, 2022

B.Sc. Computer Engineering *(conferred)*
National University of Computer & Emerging Sciences, Lahore, Pakistan August, 2014

PROFESSIONAL EXPERIENCES

Doctoral Research Fellow September, 2017 *to* Present
Boston University, Boston MA, USA

Senior Software Development Engineer August, 2016 *to* July, 2017
Mentor Graphics, Lahore, Pakistan

Software Development Engineer December, 2014 *to* July, 2016
Mentor Graphics, Lahore, Pakistan

AWARDS

Best Paper Award
IEEE International Conference on Blockchain and Cryptocurrency May, 2020

Cum Laude & Silver Medal
National University of Computer & Emerging Sciences, Lahore, Pakistan August, 2014