Increasing the robustness of the Bitcoin crypto-system in presence of undesirable behaviours

THIBAUT LAJOIE-MAZENC

Master's Thesis at CSC
Supervisor: Mads Dam
Examiner: Johan Håstad
Principal: Emmanuelle Anceaume (CNRS UMR6074 - IRISA, France)

Abstract

Decentralised cryptocurrencies such as Bitcoin offer a new paradigm of electronic payment systems that do not rely on a trusted third-party. Instead, the peers forming the network handle the task traditionally left to the third-party, preventing attackers from spending twice the same resource, and do so in a publicly verifiable way through Bitcoin's main innovation, the blockchain. However, due to a lack of synchrony in the network, Bitcoin peers may transiently have conflicting views of the system: the blockchain is forked. This can happen purely by accident but attackers can also voluntarily create forks to mount other attacks on the system.

In this work, we describe Bitcoin and its underlying blockchain protocol; we introduce a formal model to capture the normal operations of the system as well as forks and double-spending attacks. We use it to define Bitcoin's fundamental properties in terms of safety, liveness and validity.

We present the current state of the system: first, we analyse some of the most prominent works that academia has produced between 2008 and 2016, as well as some promising leads to improve the system; then, we use the results of a measurement campaign to show that the size of the network is relatively stable because join and leave operations compensate each other, and that blocks propagate to most of the network in a matter of seconds. We further compare our results to those usually accepted by the community.

We introduce a Bitcoin network simulator that we have implemented and present the experiment we have performed to validate it. Finally, we propose a modification to Bitcoin's operations that can prevent double-spending attacks and forks without giving up on its main ideological principles, decentralisation and the absence of source of trust.

Acknowledgements

I would like to express my gratitude to my principal, Emmanuelle Anceaume (PhD), for tutoring me during this thesis. She was not alone in this task, as Romaric Ludinard contributed from the definition of the problem till the end, and his contribution was appreciated as well.

Despite the distance, Mads Dam supervised me from KTH and I thank him for that. I also want to thank my examiner, Johan Håstad.

The experiment conducted on the Bitcoin network would not have been possible without the help of the technical department of INRIA Rennes Bretagne Atlantique: the staff helped me run it despite the load it put on the local network.

I would like to thank the people who accepted to review my report and have drawn my attention on many details that I would have missed otherwise. Finally, this experience would have been far less enjoyable for me without the friendliness of everyone at the lab: Bruno, Gilles, Laurent, and many others.

Contents

Chapter 1

Introduction

1.1 Motivation

In most parts of the world nowadays, currencies are controlled by a central organisation. Whether they are governments or banks may depend on the place, but that actually changes very little to the matter at hands: people are required to trust those third-parties to behave properly when they want to use money. Indeed, many things can go wrong: the third-party can create money (decreasing its unit value), and it becomes even more powerful when money gets virtualised through the use of credit cards and cashier's cheques. When that happens, banks can decide to add or withdraw any number from anyone's account; should they go bankrupt, all those trusting them with their savings would also end up facing a situation.

These few examples are but scratching the surface of an important debate in security and networking: should any system be designed as centralised, requiring a trusted third-party, or decentralised and based on a trustless model? Both cases have scenarios to back them up: a large company network would likely be better off as a centralised system because the source of trust is embedded in the company hierarchy and centralisation typically is more efficient as users can ask the third-party to shed some light on any doubtful situation. On the other hand, structures without a clear hierarchy usually have no actor universally accepted as a trusted party. Thus, a crucial question to ask when choosing between the two models is "can we trust some people to act for the greater good rather than their own?", or "is the trusted third-party trustworthy?" when a system has already been set up.

In 2008, the banking system collapsed in the subprime mortgage crisis. While most banks powered through it, it made many reconsider the question: can people really trust anyone to keep their money and generate some more out of it without being overly greedy and taking inconsiderate risks while doing so?

The same year, Bitcoin [Nak08], also known as the first successful decentralised electronic currency, was created as a negative answer to that question. Its goal was to provide the fundamental properties expected from any currency while being built on a trustless model. These properties are the following: first, no one should be able

to use someone else's money without his or her consent[1]; this is equivalent to the usual notion of *authentication*. Then, no one should be able to deny having made a transaction after it happened: it must ensure *non-repudiation*.

In this thesis, we study Bitcoin and its formal guarantees as regards these properties, especially non-repudiation because it is known to have theoretical vulnerabilities as regards it. In particular, the possibility of double-spending attacks is a source of concern: it is currently possible to repudiate transactions by sending the same coins to two different people; only one of them will get the funds, and the system will behave as if the other one had never received anything. Currently, as a precaution, people are advised to wait an average of 60 minutes before considering a transaction accepted by the network: this prevents the use of Bitcoin in many daily expenses, cups of coffee being regularly used as examples where this waiting period is unthinkable. Thus, the goal of this work is to improve Bitcoin's resilience to double-spending attacks.

This document comprises three main parts. First, Chapter 2 describes Bitcoin and its mode of operation, in order to build the formal model needed to define what Bitcoin guarantees about the evolution of the system. Then, Chapter 3 describes the current state of the Bitcoin ecosystem, both from the academic point of view through a survey of the most major papers that have been published on the topic over the years and as a system via the presentation of an experiment that was conducted during this work. Finally, Chapter 4 deals with ways to improve Bitcoin.

Additionally, we describe in Appendix A the data structures defined by Bitcoin; in Appendix B, the scripts used in transactions inputs and outputs; in Appendix C, the networking protocol followed by the reference client, Core v0.12.1; and in Appendix D, the structure of the messages described by the same protocol. Finally, Appendix E is a glossary listing all the technical terms defined or redefined in the context of Bitcoin; only some acronyms have been left out, such as those from the TCP/IP stack for the networking aspects.

During this work, we have submitted two papers: Safety Analysis of Bitcoin Improvement Proposals [ALLS16], which has been accepted, and Handling Bitcoin Conflicts Through a Glimpse of Structure [LAL], whose acceptance notification is due on November 18$^{\text{th}}$.

1.2 Notations

This section groups all the notations used throughout this document; it is intended as a quick reference and some may not make sense before being introduced in the body of the document.

[1] As far as this work is concerned, it is not the currency's role to ensure that no one gets anyone else's consent through coercion.

`0xA` Number whose hexadecimal representation is `A`. `0x` may be omitted *e.g.* for hashes;

160-hash RIPEMD-160(SHA-256($\cdot$));

256-hash SHA-256(SHA-256($\cdot$));

$\mathcal{A}$ All accounts ever created;

Alice, Bob, Carol Three nodes;

$\mathcal{B}_p$ Blockchain of store p;

$\mathcal{B}^{(b)}$ Blockchain rooted by block b;

$\mathcal{B}$ Blockchain of the whole network rooted by G_0;

$\mathcal{B}$ All well-formed blocks ever issued;

Core Version 0.12.1 of Bitcoin Core, the reference client.

$c(b)$ Content of block b, *i.e.* the list of transactions it contains;

David, Frank, Gina Three users and, by extension, their respective wallets;

f Maximum number of malicious nodes in Π;

G_0 Genesis block accepted by all the nodes in Π;

$h(\cdot)$ Function computing the 256-hash of an object;

I_T Input set of transaction T;

L_b Confirmation level of block b;

L'_b (Simplified) pseudo-confirmation level of block b;

O_T Output set of transaction T;

$\mathcal{P}_p$ Mempool of store p;

$\mathcal{P}(\cdot)$ Ancestors of a transaction or an account;

$p(b)$ Parent of block b;

s_i Solution of input i to the challenge χ_{o_i};

$\mathcal{T}$ All well-formed transactions ever issued;

$\mathcal{T}^*$ All well-formed non-coinbase transactions ever issued;

$\mathcal{V}_p$ Local view of store p;

$v(a)$ Value of account a;

W'_b Weighted pseudo-confirmation level of block b;

z Deep confirmation threshold;

z_0 Minimum length difference between two branches to prune one out of the blockchain;

$z_{\textbf{coinbase}}$ Deep-confirmation threshold for coinbase transactions;

Π Set of nodes present in the system;

$\rho_{\mathcal{B}}(\cdot)$ Minting scheme of blockchain $\mathcal{B}$;

ϕ_T Fee of transaction T;

χ_o Challenge of output o;

ζ Minimum weight difference between two branches to prune one out of the blockchain;

$\omega_{\mathcal{B}}(b)$ Weight of block b in the context of blockchain $\mathcal{B}$;

$T \triangleleft a$ Transaction T is conflictual because of account a;

$T \bowtie T'$ Transactions T, T' are conflicting;

$\equiv$ Equivalence operator between objects referring to the same coins;

$||$ Concatenation operator.

Section 4.2 additionally defines the following specific notations:

$\mathbf{PK}_{\mathcal{I}}$ Public key of identity $\mathcal{I}$;

$t_{\mathcal{I}}$ Time stamp of identity $\mathcal{I}$;

$\nu_{\mathcal{I}}$ Nonce of identity $\mathcal{I}$;

β Maximum depth of the blocks included in a new identity;

γ Target for the identity-generation process;

Δ Lifetime of identities;

ϵ Minimum difference between $f/|\Pi|$ and $1/3$;

π. Referee of a transaction, block or input.

Finally, we use the following notations in the appendix:

$\mathbf{cmpct}(\cdot)$ Length of an integer stored as a compact size unsigned integer;

$\mathcal{S}$ The set of licit encodings in 4-byte long base 256 scientific notation.

Chapter 2

What is Bitcoin?

2.1 Overview

Bitcoin was introduced in 2008 through a white-paper [Nak08] and has, since then, generated a lot of interest from several scientific communities, related to mathematics and computer science as well as economics, a number of businesses, hackers and Open-Source developers, and also national agencies. Thus, on July 3rd, 2016, a Google Scholar search on the word "bitcoin" over English pages, excluding patents and citations, returned "about 6690 results"; Bitcoin Stack Exchange [B.SE] advertised 22 517 registered users with 7376 visitors per day; and, finally, the FBI [FBI12] is but one on a long list of US agencies and bureaus that have looked into Bitcoin for several legality-related reasons such as money laundering and black market transactions, of which Silk Road, the "eBay of drugs", is a well-known example [Tra14].

Despite this, what Bitcoin actually is and does is not quite clear, hence a large vulgarization effort supported among others by books [BW14; NBFMG16] and the Bitcoin community over the Internet [B.SE; BF].

This section presents the Bitcoin protocol, as well as the cryptographic primitives it uses. Subsequently, Section 2.2 formalises what Bitcoin ensures while Section 2.3 lists some of the most well-known attacks that can target Bitcoin and its network along with their impact and remedies.

2.1.1 High-level view

In 2008, Satoshi Nakamoto, a pseudonymous author[1], published a white paper describing a way to create, distribute and manage a currency that does not rely on a trusted third party such as banks [Nak08]. The paper focuses on the major aspect of Bitcoin's data structure, the blockchain. An implementation of the system was released shortly after under the name Bitcoin Core [Core]. In the remainder of this document, we simply call Core the version 0.12.1 of Bitcoin Core, which was the lat-

[1]Or group thereof: his (or her/their) true identity remains unknown as of November 2016. Whenever necessary, we will assume that Nakamoto is a single male person.

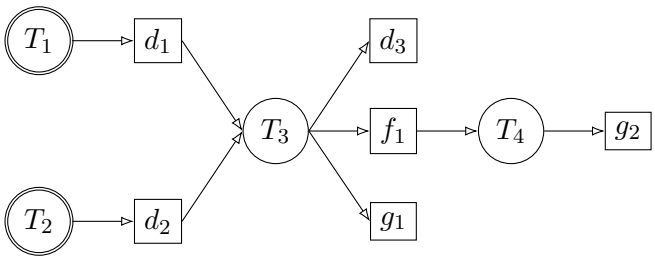

Figure 2.1: Example of transaction graph. Coinbase transactions are represented as double circles, regular transactions as circles, and accounts as rectangles. Here, David creates T_3 to send the funds he received in T_1 and T_2 to Frank on f_1 and Gina on g_1; he sends his change on a new account d_3. Then, Frank creates T_4 to send funds to Gina on g_2. Accounts d_3, g_1 and g_2 are UTxOs.

est and most used Bitcoin client during most of this work; the version number may be provided for emphasis or when describing other versions, such as Core v0.13.0 which was released during this work. We denote by Alice, Bob, and Carol three nodes of the Bitcoin network, and by David, Frank, and Gina three users of the system.

Bitcoin's goal is to provide a fully decentralised currency which resists counterfeiting attempts: units of currency, called bitcoins ("coins") or satoshis for their smallest division, cannot be created *ex nihilo* outside of the protocol. Just as bills, bitcoins have no intrinsic value: they can be transferred and exchanged at a value defined by the market. Transferring coins is done with *transactions*: the sender takes a list of input accounts, proves that she owns them, and sends their content to a list of output accounts.

A way to model this is through a directed and acyclic transaction graph: the system mints coins through special transactions called *coinbase transactions*. Afterwards, coins move from accounts used as inputs by transactions to accounts used as outputs. An account created as a transaction output that has not served as an input yet is called an unspent transaction output (UTxO). Figure 2.1 depicts a toy example.

So far, this process is similar to what happens when using a credit card. The difference is that with a credit card payment, the receiver can wait for a central authority such as Visa to confirm that the operation was successful, whereas no such central authority exists in Bitcoin. Instead, the system relies on a peer-to-peer network. Whenever a node receives a transaction, it can verify that this transaction is consistent with the history of the system, *i.e.* that the input accounts are indeed sufficiently funded to wire the advertised amounts to the output accounts.

To perform this verification operation, each node must keep a ledger recording the state of the system. Two natural ways would be to maintain a mapping between accounts and their associated value, or to record all the operations that altered the

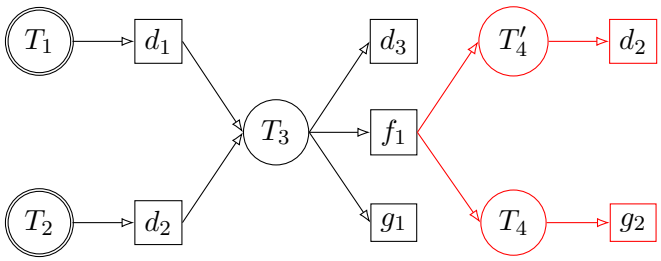

Figure 2.2: Example of double-spending attack where Frank uses transactions T_4 and T'_4 to send the same coins to David and Gina: different nodes will have different views of the ledger.

state of the system: the transactions. Both approaches have upsides and downsides, and Bitcoin chose the second one as it allows nodes to join the system at any time and still verify the validity of the entire history of the system without requiring to trust anyone in the network. Nodes can then compute and maintain the set of UTxOs as the state of the system, *i.e.* the list of accounts that can be used as transaction inputs.

A pitfall of this distributed ledger is that it requires a high level of synchronisation between nodes to maintain consistency, or else Alice could receive a transaction stating that David spends some coins before receiving the one sending them to him in the first place; in this scenario, she would reject a valid transaction. Much worse, part of the network could receive a transaction stating that Frank sends coins to David (probably in exchange for a service) while some other nodes would receive another transaction stating that he sends the exact same coins to Gina. Each part of the network would accept the first transaction it receives and reject the other one; what should happen when either David or Gina tries to spend the coins received from Frank? This is called a *double-spending attack*; Figure 2.2 shows an example of double-spending attack and Section 2.3.1 describes it in more detail.

Thus, Bitcoin needs a synchronisation mechanism to make sure that the nodes' view of the distributed ledger are consistent. It implements it with so-called *blocks*. A block is an ordered list of transactions, set in a specific history by linking to a *parent* block, produced at a slow rate. This is achieved by requiring blocks to include a *proof of work (PoW)* which ensures that the mean time needed by the network as a whole to generate a block remains 10 minutes despite the fluctuations of the network. Section 2.1.2 describes Bitcoin's PoW mechanism. All that is needed for now is that this process is random, competitive, and difficult. Thus, only certain peers try to generate blocks: the so-called *miners*. This name comes from an analogy with gold miners: mining, be it gold or blocks, requires efforts but when the miner successfully *finds* respectively a nugget or a block, she can make a profit out of it.

Indeed, since block generation is essential to Bitcoin as it is required to update the ledger, successful miners are awarded a prize: peers can only emit coinbase

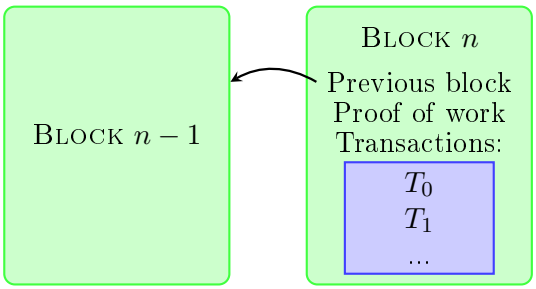

Figure 2.3: Structure of a block: T_0 is a coinbase transaction.

transactions as part of a block, with a limit of one per block. Despite not having input accounts, a coinbase transaction sends some coins to an account chosen by the miner. Those coins have two distinct origins. First, Bitcoin uses coinbase transactions to mint coins. Initially, it minted 50 coins per block; this amount is halved every 210 000 blocks, which corresponds in average to four years. Thus, block 420 000, which was the first to mint 12.5 coins, was found on Saturday, July 9$^{\text{th}}$, 2016 at 16:46:13 GMT. This halving rule yields an upper bound of 21 million bitcoins in circulation. Then, to encourage miners to include their transactions in blocks, users pay *fees* when they send funds. The fee is chosen by the sender of a transaction: it corresponds to the coins that are taken from the input accounts and not sent to any output one. Miners take the fee of each transaction they include in their blocks and add them all to the output of their coinbase transactions. Fees are intended to take the role of minting in the block reward as the latter slowly decreases and Bitcoin's volume of transactions increase. Figure 2.3 shows the general structure of a block.

This yields a sequential ordering of all the transactions recorded in the ledger: each block orders the transactions it contains, and blocks are ordered by their chaining to the one they see as their direct predecessor in the history of the system; from this chaining process also derives the name of Bitcoin's ledger: the *blockchain*. However, since mining is a random process and communications are not instantaneous, problems may arise. Specifically, nothing prevents two miners from finding two blocks that both link to the same parent. When this happens, the blockchain loses its linearity and adopts a tree structure: the chain is *forked*. The implications of such situations are described in Section 2.3.2. In short, each node selects the longest branch it has fully received as its main branch, *i.e.* what it considers to be the only valid history of the system. A fork is resolved when a branch is sufficiently longer than its competitors to appear longer to all nodes in the network despite communication delays. This usually happens quickly: even if the network is evenly distributed between two conflicting branches of equal length, it is very unlikely that miners on each branch find a block approximately at the same time because of their slow generation rate, and even more so several times in a row. Figure 2.4 shows the general structure of the blockchain and illustrates forks.

In order to make sure that a transaction confirmed in the blockchain will not be

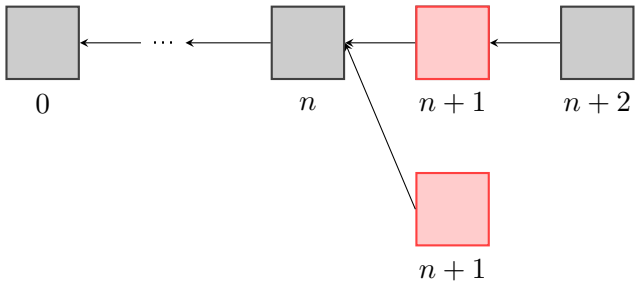

Figure 2.4: Structure of the blockchain when a fork arises. Each square represents a block.

invalidated by a fork containing a conflicting transaction, Bitcoin recommends its users to consider a block as fixed in the history of the system only when five other blocks have been found after it on its branch. This figure is derived from the fact that an attacker with less than 10 % of the total computing power will not be able to fork the blockchain before a transaction was injected in it and create a chain of size at least six before the rest of the network does with probability more than 0.1 %; this would effectively revert the transaction and make it permanently invalid [Nak08].

Finally, Bitcoin uses a flooding mechanism to propagate information: when a node receives a valid message, *i.e.* one that is consistent with its view of the state of the system, it sends it to each of its neighbours. Most nodes use a 3-way mechanism: instead of directly sending data, they send inventory messages to let their neighbours know that they can transmit them some information; whenever they receive such a message and do not know the advertised data, they ask the sender to send it. Thus, all nodes eventually receive every transaction or block that does not conflict with any local view.

With this system, Bitcoin creates a trusted third-party in a trustless network: the blockchain. Indeed, as long as attackers control less than half of the computing power, the longest chain will be dominated by honest miners. The adversary will not be able to go back to an arbitrary point in time, rewrite the history from there and catch up with the honest chain. This property, its verifiability, and the limited power given to a miner finding a single block make it a shared source of trust in a possibly adversarial network for as long as the attacker's power is sufficiently limited.

2.1.2 Supporting cryptography

This simple view of Bitcoin already highlights two crucial needs for cryptography: proofs of ownership and proofs of work (PoW). Bitcoin uses public key cryptography for the former, and hash functions for the latter.

When selling goods to David, Gina creates an Elliptic Curve Digital Signature Algorithm (ECDSA) [JMV01] key pair and asks David to sends the funds to the public key. That way, when she decides to use the account, she can simply sign the transaction using the private key, and every node will be able to verify the

signature with the public key that was included in David's transaction. That way, she proves that she owns the coins. Simply put, it corresponds to asking a bank wire to a specific account, and then using the password previously agreed upon with the corresponding bank to unlock the funds, with neither a trusted bank nor a globally accepted password. Appendix B describes the language used to define transactions outputs and inputs. In this work, we only consider signatures as proofs of ownership.

For the block generation process to work as intended, the PoW scheme must satisfy four properties:

1. the task must be computationally hard;
2. the difficulty must be parametrisable: the task must not be solvable too quickly, it should take ten minutes on average to solve it independently of the total computing power used by the network;
3. the process must be memoryless, *i.e.* changing the tentative block should not have an impact on the expected remaining time before finding a solution, or else miners would not try to include the newest transactions they receive and they might continue working on a block even after they have received a conflicting one if they know they are close to solving the task;
4. the validity of a solution must be easily verifiable.

These properties are met by random one-way functions [Bac97; DN92]: computing the image of an input is easy (Item 4), but finding a pre-image of an output is hard (Item 1). Additionally, making mining a random process with a low success probability satisfies Item 3: drawing a random number is quick and easy, and the output of a sufficiently random function will change in a completely unpredictable way whether the input is barely changed (incrementing a nonce) or heavily modified (completely changing the block to acknowledge a newly received one). This makes Item 2 easy to satisfy: it suffices to adapt the success probability to the rate at which random numbers are drawn to make the rate at which solutions are found constant.

Bitcoin has opted for the 256 bit version of Secure Hash Algorithm (SHA) 2 [FIPS180-4] (*SHA-256*) as its random one-way function. Though not perfect, it seems to constitute, as an unbroken hash function, a sufficiently good approximation as of November 2016. The success probability is tuned by setting a *target*: the PoW of a block b is valid if and only if SHA-256(b) $\leqslant$ target. This can easily be implemented via a feedback mechanism: when blocks are found too often, the target is decreased, and conversely. The *difficulty* is a more human-readable parameter, inversely proportional to the target: it estimates the amount of work needed to generate a valid PoW. Thus, when blocks are found too often and the target is decreased, the difficulty is increased. Bitcoin's target is recomputed every 2016 blocks (approximately two weeks) as per the following equation, where expected is equal to two weeks and real to the time the system took to generate the last 2016

blocks:

$$\text{new\_target} = \text{target} * \max(\frac{1}{4}, \min(4, \frac{\text{real}}{\text{expected}})).$$

Though this simple description gives an overall good idea of Bitcoin's use of cryptography, it is neither complete nor exact. Indeed, a few more concerns have been addressed. First, SHA-256 is vulnerable to length-extension attacks, where $h(m||m')$ is computed for an unknown message m using only $h(m)$ and m' for a known vulnerable hash function h. Even though there is no clear application of this attack to Bitcoin's context, all hash functions are used in pair: hashing an input to 256 bits is done through SHA-256d, applying twice SHA-256; we call the result of this operation a *256-hash*.

Then, it is theoretically possible for an attacker to develop a way to compute a private key from the corresponding public one. Since transactions are stored in the blockchain, attackers could roam through the UTxOs, crack the private keys unlocking them and create transactions to send the funds to keys under their control. Though no such attack currently exists in the literature for 64-bit ECDSA key pairs, the possibility has been mitigated by adding one more step: instead of public keys, transactions send their outputs to hashes of public keys. The most common hash function used in transactions is the 160 bit version of RACE Integrity Primitives Evaluation Message Digest (RIPEMD) [DBP96] (*RIPEMD-160*) and, to prevent length-extension attacks, it is actually applied on the output of SHA-256; we call the result of this operation a *160-hash*. Then, it suffices to disclose the public key along with the signature and verifying the proof of ownership consists in checking that the former corresponds to its advertised hash, and the latter using the former. It is furthermore recommended never to send funds twice to the same key, to make sure that no UTxO is locked by a public key that has been disclosed.

Finally, the PoW of a block is not computed using the complete list of transactions to avoid encouraging miners to mine empty blocks, but on a fixed-size placeholder, the root of the Merkle tree [Mer88] of the block. Appendix A.4 describes the construction of this tree.

2.2 Formal model for Bitcoin

This section defines a formal environment in order to explicitly derive the guarantees that Bitcoin provides to its users. Thus, it follows the standardization effort initiated by Anceaume *et al.* [ALLS16]. Given that Appendix A describes most concepts from an implementation point of view, this section focuses on a high-level modelling. We first present our assumptions about the network and then the model we propose for Bitcoin.

2.2.1 General assumptions

In order to define the composition of the network, one first needs to specify what its participants do. Thus, we rely on Antonopoulos's typology [Ant14] as follows.

Definition 1 (Roles of Bitcoin peers)
A Bitcoin peer can assume any combination of the following roles:

1. *A* (Bitcoin) router *is an entity of Bitcoin's peer-to-peer network: it maintains a dynamic set of neighbours with which it exchanges data (blocks, transactions);*
2. *A* miner *tries to solve PoWs to find blocks;*
3. *A* (blockchain) store *maintains a local copy of the blockchain, along with a mempool;*
4. *A* wallet *manages a user's keys, tracks the corresponding UTxOs and helps her create and sign transactions.*

In our model, we define a node *as a peer assuming at least the first three responsibilities; it may or may not, additionally, be a wallet.*

As per [LAL], we assume a network made of a large, finite but unbounded, set Π of nodes whose composition may change over time. Each node of Π has identical networking and computational capabilities. We define an *honest* node as one that follows the protocol specified by Core. A *Byzantine* (or *malicious*) node will do anything it can to disturb the execution of the protocol (including possibly following it whenever that benefits the adversary). Finally, a *rational* node will follow the protocol but will make any possible choice based on its self-interest; thus, it will not withhold information but may for example give priority to a transaction sending funds to itself over a conflicting one that it had received sooner but sends funds to someone else. We assume that at any time, an upper-bounded proportion of the nodes in Π are Byzantine and under the control of a single adversary. Because we focus on financial crypto-systems we consider that the rest of Π is made of rational nodes rather than honest ones. Thus, it is important that algorithms designed in this setting include incentives to encourage proper behaviour.

Communications between nodes and local computations are assumed to be both upper-bounded by constants unknown to them. Additionally, the drift between local clocks is upper-bounded. Note that such an assumption conforms with Bitcoin's usage of time stamps. This corresponds to a partial synchrony model [DLS88].

Finally, we assume that the cryptographic primitives used by Bitcoin are safe (forging signatures and finding hash collisions or pre-image are all impossible for Bitcoin's computationally bounded nodes).

2.2.2 Bitcoin model

This section follows a bottom-up approach: starting from the most basic object, it provides definitions for all the elements related to Bitcoin relevant to this work until reaching a formalisation of the following properties: Bitcoin guarantees that eventually, all transactions that are not involved in conflicts are accepted by all peers and that there is no money counterfeiting.

Let us start with Bitcoin's most low-level objects: inputs and outputs.

Definition 2 (Outputs, inputs, transactions, accounts and fees)
These elements are respectively defined as follows:

1. *an* account *a is a set of coins, whose total value is denoted $v(a)$. We denote by $\mathcal{A}$ the set of all accounts ever created in the system;*
2. *an* output *o is an account $a_o \in \mathcal{A}$ along with a challenge χ_o: to spend the former, the latter must be solved. By extension, $v(o) = v(a_o)$;*
3. *an* input *i is a pointer to an output o_i, and a solution s_i to χ_{o_i}. By extension, $v(i) = v(o_i)$;*
4. *a* transaction *T is a pair of sets: its inputs I_T and its outputs O_T. The term $\phi_T = \sum_{o \in O_T} v(o) - \sum_{i \in I_T} v(i)$ is called the transaction* fee*;*
5. *a* coinbase transaction *T_0 is a transaction whose input set I_{T_0} is empty.*

These definitions use the notion of *coins*, which has not already been defined. To avoid splitting a single coin in smaller units, the most convenient definition here is that of satoshis, the smallest unit of currency, rather than their usual meaning of bitcoins. They are naturally indexed by their order of introduction in the system and can thus be uniquely identified. See Appendix A.2 for more details. Coinbase transactions are used by the system to mint new coins through the mining process. Though accounts have no actual existence in Bitcoin, they provide for the following useful notations:

Definition 3 (Cross-type operations)
With $a, a' \in \mathcal{A}$, o, o' outputs and i, i' inputs, we define the following equivalence relation:

$$\begin{aligned} a \equiv a' &\Leftrightarrow a \ = a', \\ o \equiv a' &\Leftrightarrow a_o \equiv a', \\ i \equiv a' &\Leftrightarrow o_i \equiv a'. \end{aligned}$$

By extension, with $a \in \mathcal{A}$ and transaction T, we extend the membership relation as $a \in I_T \Leftrightarrow \exists i \in I_T, a \equiv i$ and similarly for O_T. We redefine intersection and union of sets $S \in \{I_T, O_T\}, S' \in \{I_{T'}, O_{T'}\}$ for transactions T, T' as follows, where x can indifferently be an account, an input or an output:

$$\begin{aligned} x \in S \cap S' &\Leftrightarrow \exists s \in S, \exists s' \in S', s \equiv s' \equiv x \\ x \in S \cup S' &\Leftrightarrow (\exists s \in S, s \equiv x) \vee (\exists s' \in S', s' \equiv x) \end{aligned}$$

Let us now restrict our model to objects of interest.

Definition 4 (Well-formed transactions, outputs and inputs)
We define the well-formed *property for transactions, outputs and inputs as follows:*

1. *a transaction T is said to be* well-formed *if and only if either one of the following holds:*

   a) *it is a well-formed coinbase transaction;*

   b) *all of the following holds:*

      i. $|I_T| \geqslant 1$;
      ii. $\forall i \in I_T, i$ *is well-formed;*
      iii. $\forall o \in O_T, v(o) \geqslant 0$;
      iv. $\forall T' \in \mathcal{T}, \forall o \in O_T, \forall o' \in O_{T'}, o \not\equiv o'$;
      v. $\phi_T \geqslant 0$;

   We denote by $\mathcal{T}$ the set of well-formed transactions that have been issued in the system. We denote by $\mathcal{T}^*$ its restriction to non-coinbase transactions.

2. *an output o is said to be* well-formed *if and only if $\exists T \in \mathcal{T}$ such that $o \in O_T$ and χ_o admits at least one solution;*

3. *an input i is said to be* well-formed *if and only if o_i is well-formed and s_i correctly solves χ_{o_i}.*

Objects that are not well-formed cannot propagate in the Bitcoin network because all rational nodes will reject them, and can only exist transiently as part of denial of service (DoS) attacks trying to exhaust a node's computing power or memory in repeated validity checks. Note that Definition 4 is incomplete as well-formed coinbase transactions are tackled by Definition 9. From this point forward, only well-formed inputs, outputs and transactions will be considered; through this restriction, accounts are defined as by Anceaume *et al.* [ALLS16].

A notion that is useful in the following is that of ancestors:

Definition 5 (Ancestors of transactions and accounts)
Let $T \in \mathcal{T}$ and $a \in \mathcal{A}$. We denote by $\mathcal{P}(T)$ and $\mathcal{P}(a)$ the sets of ancestors of T and a, defined respectively as:

$$\mathcal{P}(T) = \bigcup_{\{T' \in \mathcal{T} | O_{T'} \cap I_T \neq \emptyset\}} (\{T'\} \cup \mathcal{P}(T')),$$

$$\mathcal{P}(a) = \bigcup_{\{T' \in \mathcal{T}^* | \exists T \in \mathcal{T},\, a \in O_T \wedge T' \in \mathcal{P}(T) \cup \{T\}\}} I_{T'}.$$

If $T \in \mathcal{T}$ is a coinbase transaction, we have that $\mathcal{P}(T) = \emptyset$.

Note that in the definition of $\mathcal{P}(a)$, there exists at most one $T \in \mathcal{T}$ such that $a \in O_T$ by Definition 4. The initialisation of the induction derives from the fact that the input set of coinbase transactions is empty. The set of ancestors of any transaction or account is finite, because the system only introduces coins through coinbase transactions: for any account, it is possible to follow the trail of accounts to the time each of the coins it contains was minted.

Definition 6 (Double-spending and conflict situations)
$a \in \mathcal{A}$ *is said to be in a* double-spending situation *if and only if* $\exists T, T' \in \mathcal{T}, a \in I_T \cup I_{T'}$.

Furthermore, $a' \in \mathcal{A}$ *is said to be* conflictual *if and only if* $\exists a \in \mathcal{P}(a') \cup \{a'\}$ *such that* a *is in a double-spending situation. Conversely, non-conflictual accounts are said to be* conflict-free.

By extension, $T \in \mathcal{T}^*$ *is said to be* conflictual *if and only if* $\exists a \in I_T$ *such that* a *is conflictual. For each such* a*, we denote by* $T \triangleleft a$ *that* T *is conflictual because of account* a.

Transactions $T, T' \in \mathcal{T}^*$ *are said to be* conflicting *if and only if* $\exists a \in \mathcal{A}, T \triangleleft a \wedge T' \triangleleft a \wedge T \notin \mathcal{P}(T') \wedge T' \notin \mathcal{P}(T)$. *This is denoted* $T \bowtie T'$.

Finally, $T \in \mathcal{T}^*$ *is said to be* conflict-free *if and only if* T *is not conflictual.*

Informally, two transactions are said to be conflicting whenever they both use the same coin without any of the two being a descendant of the other.

The special case of conflictual coinbase transactions is handled in Definition 10. Note that a transaction may be conflictual because of several distinct accounts: this is a many-to-many relation.

Definition 7 (Conflict-free transaction according to Anceaume *et al.*)
Anceaume et al. *[ALLS16] consider transaction* T *conflict-free if and only if* $\forall a \in I_T, a$ *is not in a double-spending situation and* $\forall T' \in \mathcal{P}(T), T'$ *is conflict-free.*

Property 1 (Equivalent definitions of conflict-free transactions)
Definitions 6 and 7 are equivalent for conflict-free non-coinbase transactions.

Proof. Let us first show that Definition 6 is a sufficient condition for Definition 7. Let $T \in \mathcal{T}^*$ and let some $a \in I_T$ be conflictual. Then either a is in a double-spending situation and T is not conflict-free as per Definition 6, or $\exists a_0 \in \mathcal{P}(a), a_0$ is in a double-spending situation. By the definition of $\mathcal{P}(a)$, $\exists T_0 \in \mathcal{T}^*$ such that $a_0 \in I_{T_0}$. Thus, T_0 is not conflict-free and neither is T, as per Definition 6 hence the partial result.

Similarly, Definition 6 is a necessary condition for Definition 7: let $T \in \mathcal{T}^*$ not be conflict-free as per Definition 6. Then either $\exists a \in I_T$ in a double-spending situation and thus a is conflictual and T is not conflict-free according to Definition 7 either, or we can find $T' \in \mathcal{P}(T)$ such that $\exists a' \in I_{T'}$ in a double-spending situation and thus conflictual. This makes use of the finiteness of $\mathcal{P}(T)$. The result derives from the fact that by definition of $\mathcal{P}(T)$, $\exists T'' \in \mathcal{T}, T' \in \mathcal{P}(T'') \cup \{T''\} \wedge O_{T''} \cap I_T \neq \emptyset$: all the accounts from $O_{T''} \cap I_T$ are conflictual because T'' is conflictual. □

Now that every concept related to non-coinbase transactions has been defined, blocks can be formally defined as well, in a similar way. Along with them, coinbase transactions can be covered.

Definition 8 (Blocks and blockchain)
We define these elements as follows:

1. *A* block *b is an ordered set of transactions $c(b)$, and is either a* genesis *block or has a* parent block *$p(b)$. Block b is called a* successor *of $p(b)$. By convention, $p(b_0) = b_0$ for a genesis block b_0.*

2. *The* blockchain *$\mathcal{B}^{(b_0)}$ rooted by a genesis block b_0 is the set $\{b \text{ block} | \exists k \in \mathbb{N}, p^k(b) = b_0\}$, where $p^k(\cdot)$ is the composition of $p(\cdot)$ with itself k times ($p^0(\cdot) = \cdot$). The height of a block b in its blockchain is $\min\{k | p^k(b) = p^{k+1}(b)\}$, i.e. its depth in the tree.*

3. *A* minting scheme *$\rho_{\mathcal{B}}(\cdot) : \mathbb{N} \to \mathbb{R}_+$ is a function mapping block heights in a blockchain $\mathcal{B}$ to a positive amount of coins. By extension, for a block b of height k, $\rho_{\mathcal{B}}(b) = \rho_{\mathcal{B}}(k)$;*

4. *A* weighing scheme *$\omega_{\mathcal{B}}(\cdot)$ is a function mapping blocks to real numbers in the context of a blockchain $\mathcal{B}$.*

Though the definition of blockchain does not forbid the coexistence of several differently-rooted blockchains, we enforce that all nodes from Π use the same genesis block G_0, as is the case in Bitcoin's main network whose root is the block `000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f`; thus, in the following, we will simply denote $\mathcal{B}^{(G_0)}$ by $\mathcal{B}$. The minting scheme is the function defining the number of coins a miner can mint when she finds a block: it corresponds to Bitcoin's 50 bitcoins halved every 210 000 blocks. The weighting scheme combines Bitcoin's PoW and difficulty, without actually restraining the model to PoW-only mechanisms; it can assign a negative value when the proof is invalid (*e.g.* the block's hash is above its target) and a value depending on the proof used such as the block's difficulty for Bitcoin's PoW otherwise.

Definition 9 (Well-formed coinbase transactions and blocks)
We define the well-formed *property for coinbase transactions and blocks as follows:*

1. *A coinbase transaction T_0 is said* well-formed *if and only if all the following conditions hold:*

   a) *There exists a block $b \in \mathcal{B}$ such that $T_0 \in c(b)$;*

   b) *$\forall T \in \mathcal{T}, \forall o \in O_{T_0}, \forall o' \in O_T, o \not\equiv o'$;*

   c) *$\sum_{o \in O_{T_0}} v(o) = \rho_{\mathcal{B}}(b) + \sum_{T \in c(b) \setminus \{T_0\}} \phi_T$;*

2. *A block b in a blockchain $\mathcal{B}$ is said* well-formed *if and only if all the following conditions hold:*

   a) *$\forall T \in c(b), T$ is well-formed;*

   b) *$\exists! T_0 \in c(b), |I_{T_0}| = 0$;*

   c) *$\omega_{\mathcal{B}}(b) \geqslant 0$;*

   d) *No transaction appears more than once in $c(b)$;*

e) b is either a genesis block or all the following conditions hold:

i. $\exists! b' \in \mathcal{B}, b' \neq b \wedge p(b) = b'$;

ii. $\forall T \in c(b), \forall k \in \mathbb{N}, T \notin c(p^k(b))$;

iii. $\forall T \in c(b), \forall k \in \mathbb{N}, \nexists T' \in c(p^k(b)),\ T \bowtie T'$;

We denote by $\mathcal{B}$ the set of all well-formed blocks that have been issued in the system.

Informally, forks are to blocks what double-spending attacks are to transactions; a fork consists in two blocks having the same parent just as a double-spending attack consists in two transactions having the same input. Formally,

Definition 10 (Conflicting blocks and coinbase transactions)
Blocks $b, b' \in \mathcal{B}$ are said to be conflicting *if and only if $\exists k, k' \in \mathbb{N}, p^k(b) = p^{k'}(b') \wedge \forall \kappa, \kappa' \in \mathbb{N}, p^{\kappa}(b) \neq b' \wedge p^{\kappa'}(b') \neq b$. This is denoted $b \bowtie b'$ as well.*

A block is conflictual *if there is another one such that the pair is conflicting. Otherwise, it is* conflict-free.

A coinbase transaction T_0 is conflictual *if and only if its block b is conflictual. In that case, it is* conflicting *with the coinbase transaction T_0' of any block b' such that $b \bowtie b'$ and any transaction T' such that $T_0' \in \mathcal{P}(T')$.*

The issue with this definition of conflict for coinbase transactions is that as soon as a fork arises, all subsequent blocks, and thus coinbase transactions, are conflictual: eventually, most accounts will become conflictual. Indeed, transaction fees make coins pass several times through coinbase transactions during their (infinite) lifetime. To prevent this, we need the ability to consider a fork resolved. Then, we will be able to prune the losing branch out of the blockchain: all the blocks of the winning branch will lose their conflictual status unless they are also involved in a subsequent fork.

Definition 11 (Weighted pseudo-confirmation level of a block)
Let $b \in \mathcal{B}$. The weighted pseudo-confirmation level *of b, denoted W_b', is the quantity*

$$W_b' = \max\{\sum_{i=0}^{k} \omega(b_i) | \exists k \in \mathbb{N}, \exists b_0, ...b_k \in \mathcal{B}, b_0 = b \wedge \forall i \in [[1,k]], p(b_i) = b_{i-1}\}.$$

Definition 12 (ζ-prunable blocks, ζ-linearised blockchains, ζ-resolved forks)
Let $b \in \mathcal{B}$ such that b is conflictual. b is said ζ-prunable *for some $\zeta \in \mathbb{R}_+$ if and only if $\exists b', b'' \in \mathcal{B}$ such that all of the following conditions hold:*

1. *$b' \in \mathcal{P}(b) \cup \{b\}$;*
2. *b' and b'' have the same height;*
3. *$b' \bowtie b''$;*

4. $W'_{b''} - W'_{b'} \geqslant \zeta$;

When that happens, b is said ζ-prunable because of *(the branch of) b''.*

The ζ-linearisation *of a blockchain B, denoted $B^{(\zeta)}$, is the restriction of B to its blocks that are not ζ-prunable.*

A fork is said ζ-resolved *when all its branches but one are ζ-prunable.*

Informally, the pseudo-confirmation level of a block corresponds to the total weight of the heaviest branch it roots. As soon as a block becomes ζ-prunable, all of its descendants get the same status: it suffices to evaluate whether the two (or more) blocks that initiated a fork are ζ-prunable to determine whether the entire branch they each root is ζ-prunable. This notion is useful because a ζ-prunable branch only has a negligible probability (in ζ) of ever catching up with the branch with which it conflicts as long as the weight of a block sufficiently correlates with the amount of effort its generation required. This is the case in Bitcoin with the weight corresponding to the difficulty of the PoW. The gist of the proof is the same as that used by Nakamoto to explain why one should wait 6 blocks before considering a transaction as recorded in the blockchain [Nak08].

However, it is also very cumbersome to use because ζ must be adjusted correspondingly with the weighing scheme and the evolution of the system. Indeed, for Bitcoin's PoW, the target is regularly adjusted to account for the overall variations in the computing power of the miners: this roughly represents a ten-fold increase between late 2014 and November 2016 [BC.I]. Thus, a ζ corresponding to one hour of work in November 2016 would have required ten times as much in late 2014, which makes for a very slow conflict resolution; on the other hand, one hour in 2014 corresponds to less work than a single block two years later, and the probability that the losing branch catches up is not negligible any more.

For this reason, when the weight of well-formed blocks is piecewise constant for ranges of heights in their blockchain, the following characterisation is easier to use.

Definition 13 (Simplified pseudo-confirmation level of a block)
Let $b \in \mathcal{B}$. The simplified pseudo-confirmation level *of b, denoted L'_b, is the quantity*

$$L'_b = \max\{k+1 | \exists k \in \mathbb{N}, \exists b_0, ...b_k \in \mathcal{B}, b_0 = b \wedge \forall i \in [[1,k]], p(b_i) = b_{i-1}\}.$$

By default, the pseudo-confirmation level of a block refers to its simplified *definition.*

Definition 14 (z_0-prunable blocks, z_0-linearised blockchains, z_0-resolved forks)
Let $b \in \mathcal{B}$ such that b is conflictual. Block b is said z_0-prunable *for some $z_0 \in \mathbb{N}$ if and only if $\exists b', b'' \in \mathcal{B}$ such that all of the following conditions hold:*

1. $b' \in \mathcal{P}(b) \cup \{b\}$;
2. *b' and b'' have the same height;*
3. $b' \bowtie b''$;

4. $L'_{b''} - L'_{b'} \geqslant z_0$;

When that happens, b is said z_0-prunable because of *(the branch of) b''.*

The z_0-linearisation *of a blockchain B, denoted $B^{(z_0)}$, is the restriction of B to its blocks that are not z_0-prunable.*

A fork is said z_0-resolved *when all its branches but one are z_0-prunable.*

This second definition is much more usable in practice because weighting schemes are typically adjusted for resource-consuming mining schemes to keep the pace at which blocks are generated approximately constant. However, it becomes less safe when a weight modification (*e.g.* a target adjustement) happens during a fork because a branch can get the ability to produce more blocks with less work than its competitors, hence the necessity to fall back to ζ-linearisation when that happens.

Definition 15 ((z_0, ζ)-linearised blockchains)
The (z_0, ζ)-linearisation *of a blockchain B rooted by a genesis block G_0, denoted $B^{(z_0,\zeta)}$, is a subset of B defined as follows:*

1. $G_0 \in B^{(z_0,\zeta)}$;
2. *If* $b \in B^{(z_0,\zeta)} \wedge \exists! b' \in B, p(b') = b$ *then* $b' \in B^{(z_0,\zeta)}$;
3. *Else, if* $\exists b \in B^{(z_0,\zeta)}, \exists b' \in B, \forall b'' \in B$ *such that* $p(b') = p(b'') = b$, b'' *is* z_0*-prunable because of* b', *and with* $k = \max\{i \in \mathbb{N} | \exists b_i \in B, b'' = p^i(b_i)\}$ *we have that* $\forall i \in [[0, k]], \forall b_{i,0}, b_{i,1} \in B, b' = p^i(b_{i,0}) \wedge b'' = p^i(b_{i,1}) \wedge \omega(b_{i,0}) = \omega(b_{i,1})$ *then* $b' \in B^{(z_0,\zeta)}$;
4. *Else, if* $\exists b \in B^{(z_0,\zeta)}, \exists b' \in B, \forall b'' \in B$ *such that* $p(b') = p(b'') = b$, b'' *is* ζ*-prunable because of* b' *then* $b' \in B^{(z_0,\zeta)}$;
5. *Else,* $\forall b' \in B, p(b') = b$ *then* $b' \in B^{(z_0,\zeta)}$.

Informally, Item 2 accepts all blocks that do not introduce a fork in the linearised blockchain; Item 3 performs the z-linearisation for branches where each pair of block of the same height has the same weight; Item 4 performs the ζ-linearisation for branches where the weight function gives different values on different branches for the same height; finally, Item 5 accepts both conflicting branches when none is prunable.

As opposed to a blockchain B, none of the three restrictions defined above is of monotonically increasing size: when a fork arises, all its conflicting branches are included in the restrictions before being pruned out when they become prunable, which resolves the fork. With very high probability (in z_0 and/or ζ), a branch that has been pruned out of a restriction cannot join it back, as long as the fraction of computing power the adversary controls is sufficiently small. Both the upper bound and the exact dependency are left for future work.

It is possible to define ζ as a function of the state of the blockchain, *e.g.* to keep it roughly equal to the amount of effective computing power used by the network over

a fixed period of time; the only requirement it that any pair of conflicting blocks can be compared to determine whether a sufficient amount of work has been dedicated to extending the heaviest subchain rooted by one of them to ensure that the other one will not be able to catch up. With infrequent target adjustments, most of the forks will be handled by the z-linearisation scheme. To simplify the notations, we denote by $B^* = B^{(z_0,\zeta)}$ the linearisation of a blockchain B with (z_0, ζ) considered security parameters of the system.

We can use this linearisation to define the confirmation level of a block:

Definition 16 (Confirmation level and deep confirmation of blocks and transactions)
The confirmation level L_b *of some block* $b \in B^*$ *is defined as follows:*

$$\begin{aligned} L_b = \max(\{k+1 | \exists k \in \mathbb{N}, \exists b_0, ...b_k \in B^*, \\ b_0 = b \text{ and } b_0 \text{ conflict-free} \\ \wedge \forall i \in [[1,k]], p(b_i) = b_{i-1} \text{ and } b_i \text{ conflict-free}\} \cup \{0\}). \end{aligned}$$

b is said to be deeply confirmed *if and only if $L_b \geqslant z$. Parameter z is called the* deep confirmation threshold.

By extension, the confirmation level *of a transaction is equal to that of the block containing it, or 0 if no such block exists. Similarly, a transaction is said to be* deeply confirmed *if and only if its confirmation level is greater than or equal to z.*

The deep confirmation threshold z is also a security parameter, that should be chosen greater than z_0. Informally, once a block becomes deeply confirmed, it cannot be pruned out of the linearised blockchain by a malicious miner. Though it would be an interesting property, the confirmation level of a block is not monotonically increasing: any miner can fork the blockchain so that some of its last blocks become conflictual, which decreases the confirmation level of all the blocks. It may even transiently cause a block to lose its deep confirmation; however, in such a case, the malicious branch would fail to win the fork and the previously deeply-confirmed blocks would get this status back.

The case where a transaction is included in several blocks does not lead to it having several distinct confirmation levels: in such a situation, the blocks are necessarily conflicting (by definition of a well-formed block) and thus they all have a confirmation level of 0. This lets us solve the transaction conflicts as well: once a transaction is deeply confirmed, all those conflicting with it will with high probability never be included in the blockchain: they can thus be erased from $\mathcal{T}$ along with all those admitting them as ancestors and the corresponding accounts can as well be erased from $\mathcal{A}$.

An issue closely related to double-spending is the fast use of newly minted coins: if someone were to use the output of the coinbase transaction of a very recent block, there would be a chance for the block to be pruned, which leads to pruning the coinbase transaction as well. Thus, it could be possible to use money that will eventually never have existed. To prevent that from happening, the outputs of

coinbase transactions are locked until the transaction that created them reaches a confirmation level of $z_{coinbase}$ that should be chosen greater than z.

Since we consider a partially synchronous model, nodes may have different local views of the system.

Definition 17 (Local view)
At any time, node p only has a local view $\mathcal{V}_p = \mathcal{B}_p \cup \mathcal{P}_p$ of the system, comprising:

1. *a* local *blockchain $\mathcal{B}_p$, the restriction of the linearised blockchain B^* (rooted by G_0) to the blocks that p has received. A transaction T is said to be in $\mathcal{B}_p$ if and only if $\exists b \in \mathcal{B}_p, T \in c(b)$. By extension, it is also denoted $T \in \mathcal{B}_p$;*

2. *a* local *mempool $\mathcal{P}_p$, a pool of locally valid transactions that have not been locally confirmed yet.*

This definition uses the notion of local validity [ALLS16]:

Definition 18 (Local validity)
A node p considers a transaction T as locally valid if and only if the following properties hold:

$$\forall a \in I_T, \exists T' \in \mathcal{V}_p, \quad a \in O_{T'} \tag{2.1}$$

$$\forall T' \in \mathcal{V}_p, \quad I_T \cap I_{T'} = \emptyset \tag{2.2}$$

Relation (2.1) is the existence property: the inputs of T must unlock outputs that p has received. As we assume that all transactions are well-formed, it suffices for p to have received the transaction creating the output referred to by each input for the input to correctly unlock it. Relation (2.2) is the availability property: the inputs must not have already been spent. Anceaume *et al.* [ALLS16] include a third relation: $\forall a \in O_T, \forall T' \in \mathcal{V}_p, a \notin O_{T'}$. It corresponds to the absence of account reuse. However, assuming that there is no collision in transaction hashes, this property is always verified because each element of O_T is referred to through the tuple (T, i) where i is the index of the element in O_T. Note that there have been collisions for two pairs of coinbase transactions: the property is thus not as trivial as it may look in practice, our model simply hides it in the well-formed characteristic.

Whenever a node receives a transaction that is not locally valid because it conflicts with a subset of the mempool, it can choose either to keep the previously accepted subset or to replace it with the new transaction. Conflicting blocks are handled similarly when the two resulting blockchains have the same total weight; otherwise, the heaviest one is chosen by rational nodes as it is the most likely to survive the eventual linearisation of the blockchain.

Finally, the last concept of our model is that of local confirmation.

Definition 19 (Local and deep confirmation)
The local confirmation level *of a transaction T for a peer p is the confirmation level of T in $\mathcal{B}_p$.*

T is said locally deeply confirmed *for a peer p when its local confirmation level is above z for p. For simplicity, T is said* deeply confirmed by *p.*

Bitcoin currently uses $z = 6, z_{coinbase} = 100$, and only the ζ-linearisation method with $\zeta = 1$, from assumptions made by Nakamoto [Nak08]. However, this is unsafe because whenever a fork arises, all blockchain stores immediately choose to prune out a branch; the probability that a previously pruned out block is reintegrated in the blockchain is thus far from being negligible.

All these definitions lead to the following statement of the fundamental properties of Bitcoin, adapted from our published work [ALLS16]:

Property 2 (Bitcoin's liveness)
A conflict-free transaction will eventually be deeply confirmed by a rational node.

Property 3 (Bitcoin's safety)
A conflict-free transaction deeply confirmed by some rational node will eventually be deeply confirmed by all rational nodes at the same height in the blockchain.

Property 4 (Bitcoin's validity)
Any transaction deeply confirmed by some rational node is not conflicting with any other transaction deeply confirmed by the same node.

Property 3 ensures that rational nodes share a common prefix for the blockchain. The exact definition of "eventually" in terms of the deep confirmation threshold z and the communication delays is left for future work.

2.3 Known vulnerabilities

Despite the efforts invested in mitigating them, Bitcoin is subject to vulnerabilities. While some are inherent to peer-to-peer networks, others are more specific to the blockchain technology or to the Bitcoin protocol. This section describes the most most well-known of them. It does not mention attacks against the cryptographic primitives, which are outside the scope of this document but should not be forgotten: forging signatures and inverting hashes would be devastating to Bitcoin. However, since the primitives used are well established, it is still reasonable as of this writing to assume that they are not broken *yet*.

2.3.1 Double-spending attacks

A double-spending attack consists in sending two transactions with non-disjoint sets of inputs and getting services or goods in exchange of both while only one can be accepted in the blockchain (alternatively, one can also send two conflicting transactions, get services for one and manage to have the other, sending funds back to oneself, accepted in the blockchain).

The general idea is for David, who wants to buy services from Frank and Gina, to sign a transaction T sending funds to Frank, collect his service and, before Gina

receives T, to sign T' sending funds to her such that $I_T \cap I_{T'} \neq \emptyset$ and collect her service. Thus, only one of T, T' will be accepted in the network and all the accounts in $I_T \cap I_{T'}$ will have been used by David to acquire two services at the price of one. The reason why this attack does not apply to traditional currencies is that they include, in their electronic version, a trusted third-party. If David were to use a credit card to buy from Frank and Gina, both of them would quickly get a confirmation from *e.g.* Visa that the payment was valid. Without this centralised source of trust, no one can ensure them that they will get their due.

This attack is the reason why nodes should not consider non-deeply confirmed transactions definitely recorded in the ledger: transactions in blocks have precedence over those in the mempool and blocks may occasionally be pruned out of the blockchain. The validity property ensures that no two conflicting transactions are deeply confirmed by a single rational node but does not prevent an attacker from getting two different rational nodes to locally deeply confirm two conflicting transactions by hiding the presence of a fork to them. The prunability parameters and deep confirmation threshold must be chosen so as to limit the feasibility of such an attack, which is possible because nodes only have a limited view of the blockchain and compute confirmation levels based on this partial information.

2.3.2 Forks

Forks arise naturally because of the asynchrony of the network: the chance that a miner finds a block while another one with the same parent is propagating through the network depends on the time it takes for a block to propagate to the whole network. For example, assuming that a block is propagated simultaneously to all miners one second after it has been found, there is a probability $p \approx 1/600$ that a conflict arises (assuming a geometric distribution with a mean of 10 minutes for the mining process). This follows a geometric distribution and, in this oversimplified model, a non-malicious fork arises every 600 blocks (approximately 4 days and 4 hours) on average.

They are harmful to the system because they cause inconsistencies in the local views of the system state. Indeed, the sets of UTxOs will necessarily be different because the respective coinbase transactions create different outputs. Forks are eventually resolved because each rational store considers the heaviest chain it knows (the one with the highest cumulative difficulty) to be the valid one and, eventually, all rational miners end up working on the same branch, assuming a reasonable upper-bound on the communication delays (the lower the bound, the quicker the resolution in terms of blocks).

However, forks may also be malicious: a Byzantine miner can deliberately work on a forked branch. The most obvious benefit is that she will get the block reward of each block she finds in her branch if it wins the fork, but another one is that she can use the fork to perform double-spending attacks. Indeed, if Carol manages to fork the network between branches A and B, and to insert a transaction to Alice's wallet in A and a conflicting one sending the same coins to Bob's wallet in B, and

to have Alice accept A and Bob B (which can happen as long as the two branches have the same weight), she can have them both send her their goods before the fork is resolved.

From this derives the notion of *effective network computing power*, the computing power dedicated to extending the heaviest branch of the network, decreased by forks and propagation delays (*e.g.* using the same model as before but with a 3 s propagation time, an average of approximately $3/600 = 0.5\,\%$ of the total computing power is lost in propagation delays) [DW13].

Forks are handled by Bitcoin's liveness and safety properties as conflict-free transactions can be included in all conflicting branches.

2.3.3 Network split

Splitting the network consists in dividing the network in two or more graphs that are not connected to each other. Should that happen, the blockchains of each graph would diverge and the usual mechanism to resolve forks would not work. At the end of the attack, when the graphs manage to recreate interconnections, the graph which was the most successful in generating blocks would impose its view of the network. The main point of this attack is that the fork will last for at least as long as the split: it creates a malicious fork without consuming computing power.

The connection management implemented by Core is quite complex in order to protect the network against such attacks. First, each router establishes outbound connections, which means that an attacker would need to fill up his address manager with malicious entries[2] to prevent it from creating links crossing the split. Given the structure of the address manager, described in Appendix C.2, this is quite hard to establish. Then, the attacker would need to disrupt the connections crossing the split: this either requires controlling the network between each pair of neighbours to drop all the exchanged messages and force connection time outs, or filling up each router's set of neighbours and managing to evict those that are on the other side of the attempted split. Once again, given the randomized approach to evicting neighbours, described in Appendix C.1, this seems impractical.

All in all, though potentially devastating, network splitting attacks do not seem more feasible through the Bitcoin protocol than general large-scale attacks on the Internet or, at least, maintainable for a sufficiently long time to mount other attacks.

2.3.4 51 % attack

The 51 % attack relates to the fact that an attacker controlling more than half of the total computing power in the network would be all-powerful. Without checkpoints on the blockchain, it could go back to the genesis block, fork the blockchain and still produce a chain longer than that mined by all the other miners, thus rewriting the entire history of the system. Any transaction could become conflictual as they all rely at least on the coinbase transactions that generated their input coins, which

[2] This is called a Sybil attack: an attacker needs to create a large number of valid identities.

could all belong to conflicting blocks depending on which prefix of the blockchain the attacker preserves.

Thus, this attack does not properly fit in our model as it conflicts with the hypothesis that coinbase transactions only become accepted in the system after a period of time such that no attacker could revert them, which is precisely what the 51 % attack does. It suffices to control slightly more than half of the *effective* computing power to perform this attack: similarly to forks, any technique decreasing the effective computing power (such as disrupting communications or targeting miners with DoS attacks) facilitates it.

There is currently no specified automated protection against 51 % attacks. However, there are still two reasons why they are not performed, or not enough to be noticed. First, since February 3$^{\text{rd}}$, 2016, the computing power measured by Blockchain.info [BC.I] has never dropped below 10^{18} hash per second. This is a considerable amount, and controlling half of that is not an easy task. Then, performing this attack in too obvious a manner would probably not be profitable: it would surely lead to a hard fork, where the honest community would just separate from the corrupted network; to a drop of the price of bitcoins; to a complete abandonment of the system; or to a combination of these scenarios. This explains why our model does not take it into account: the fact that the system as a whole would be doomed if it happened combined with its impracticality in large enough networks make it a very specific issue.

2.3.5 Malicious mining

Byzantine peers can deviate from all parts of the protocol, and not only the ones related to communications. As such, there are ways for them to mine maliciously in order to increase their expected profits. Two such ways are SPV mining [BW.MP] and selfish mining.

SPV mining consists in mining on top of blocks that have not been locally validated; its name derives from the Simplified Payment Verification (SPV) mode of operation in which resource-constrained nodes do not receive or verify the full blockchain. Its simplest form is for miners to accept block b as the parent of the block b' that they are trying to build as soon as b is received, without taking the time to validate it first (and possibly to even check whether it is well-formed). This only has a low impact, as validating a block takes less than a second. However, a more elaborate form consists in listening to other sources, such as the web APIs of the biggest pools to get the hash of newly found blocks even before they propagate in the network; this can lead to a few seconds of SPV mining. The main reason it is considered an attack on the network is that it is unfair to the miners doing what they are supposed to be rewarded for, creating a clean ledger. On the other hand, it is risky because any block built on top of an invalid one is invalid as well. Finally, it has the positive side-effect of decreasing the probability of forks because it decreases the time spent on involuntarily trying to fork the blockchain.

Selfish mining [ES14a] is a more elaborate adversarial behaviour. It consists, for Alice, in keeping the blocks she finds to herself for as long as possible before releasing them in the network. Thus, she can be the only one mining on top of them for a long period of time, increasing her relative computing power. This attack fails if *i)* she keeps her blocks for too long, *ii)* a fork occurs, and *iii)* the other branch wins. If she has a broadcasting advantage (for example, by being connected to strategic points of the network), she can decrease this failure probability and keep her blocks secret even longer. Specifically, the use of parallel networks helping miners propagate their blocks (*e.g.* the Fast Relay Network [Cor16]) can help Alice win forks with great probability when the other miner(s) involved does not use them. As opposed to SPV mining, this truly is an attack in that it makes rational miners work on blocks that already have successors; thus, it bears some resemblance with communication disruption attacks.

However, none of these attacks affect Bitcoin's properties: they increase the likelihood that a block is found by a Byzantine miner but conflict-free transactions are relatively unaffected, except for the fact that it may have an impact on the time needed for them to get deeply confirmed.

Chapter 3

Bitcoin today

Since its inception in 2008, Bitcoin has changed in several meaningful ways, while keeping some of its core parts intact. Thus, the market price of coins has drastically increased [Caf16], but transactions and blocks remain mostly unchanged. In this chapter, we present a survey of selected papers relevant to our study, and we describe and analyse the results of an experiment that we performed to evaluate the state of today's Bitcoin network.

3.1 Survey of selected academic papers

Bitcoin, its blockchain structure and its decentralized model have generated a lot of interest from the scientific community. Given the number of papers published in the field over the past few years, this survey only focuses on topics this thesis deals with. Examples of topics that are not covered include privacy (of users and peers [BKP14; GCKG14]), many altcoins [ANV13], and evaluations of Bitcoin's decentralisation [GKCC14] or financial and regulatory ramifications [DF14] This survey groups papers based on the subparts of the global system they tackle: first, application-agnostic models for the blockchain are presented; then come papers presenting the results of large-scale measures performed on the Bitcoin network, followed successively by the topics of information propagation, malicious mining and double-spending attacks. Finally, we conclude this survey with papers focusing on bigger revisions of the Bitcoin protocol.

Table 3.1: Summary of the papers described in Section 3.1.

Paper	Year	Model	Feasibility
The Bitcoin Backbone Protocol: Analysis and Applications [GKL15]	2015	Fixed synchronous network with an adaptive and rushing adversary.	Application-agnostic but strong assumptions.

Table 3.1: (continued)

Paper	Year	Model	Feasibility
Analysis of the Blockchain Protocol in Asynchronous Networks [PSS16]	2016	Partially synchronous, dynamic, and adversarial network.	Application-agnostic; only studies malicious mining.
Information Propagation in the Bitcoin Network [DW13]	2013	Random graph, uniform computing power.	Partial results, may simplify DoS attacks.
Discovering Bitcoin's public topology and influential nodes [MLPG+15]	2015	Peers follow Core's undocumented management of address time stamps.	Relies on undocumented and/or patched behaviour.
On Bitcoin and Red Balloons [BDOZ11]	2012	Full propagation before mining. Forest of trees.	No implementation, double-spending vulnerability.
Tampering with the Delivery of Blocks and Transactions in Bitcoin [GRKC15]	2015	Reasonable adversarial networking and computing capabilities.	Already partially implemented.
Bitcoin: a peer-to-peer electronic cash system [Nak08]	2008	Implicit synchrony and very weak adversary model.	Founding paper with overly generalised conclusions.
Majority Is Not Enough: Bitcoin Mining Is Vulnerable [ES14a]	2014	Synchrony and absence of accidental forks.	The situation is even worse with more realistic assumptions.
Double-Spending Fast Payments in Bitcoin [KAC12]	2012	Non-mining adversary.	Based on Core v0.5.2: paying to IP addresses is no longer supported.
Have a Snack, Pay with Bitcoins [BDEWW13]	2013	Non-mining adversary, modified behaviour for the victim's router.	May harm the network if deployed at large scale.
Safety Analysis of Bitcoin Improvement Proposals [ALLS16]	2016	Partially synchronous, dynamic, and adversarial network.	(Not applicable)
Secure High-Rate Transaction Processing in Bitcoin [SZ15]	2015	Very few restrictions on the graph, weak adversary model.	Adapted for Ethereum [But14] but wasteful.
Cryptocurrencies without Proof of Work [BGM14]	2014	Stakes are distributed enough.	Preliminary design with interesting properties.

3.1.1 Blockchain models

This section groups papers providing models to study the properties of blockchains in an application-agnostic setting. To the best of our knowledge, there is no other prominent paper in the field as of this writing.

3.1.1.1 Blockchains in synchronous networks

In the Bitcoin Backbone Protocol: Analysis and Applications [GKL15], the authors present a formal model to study blockchain protocols in synchronous networks. They define the two properties of common prefix and chain quality, and use them to study different blockchain-based applications for *e.g.* solving Byzantine agreement problems.

The model assumes a network without churn that follows a protocol in rounds. At each round, each node can try the same number of nonces in the mining process and all messages are sent and received. The adversary can choose which nodes to corrupt, the only limits to the corruption being that she can only control an upper-bounded fraction of the nodes and they remain computationally bounded. Corrupted nodes can, however, receive all the messages sent in a round before any other node, define their behaviour for the round in consequence, and get all honest nodes to receive their messages before those originating from other honest nodes. They cannot tamper with the content or the delivery of an honest message but they can choose to send different messages to different nodes.

In this scenario, they show that with high probability, all honest nodes share the same prefix of the blockchain (common prefix property) and the blocks found by the adversary represent a limited fraction of the total number of blocks in the blockchains (chain quality property).

They use these two properties to build two protocols that solve binary Byzantine agreement, a problem that combines the four usual properties of consensus (termination, validity, integrity, agreement) in two that they call validity (covering termination as well) and agreement (including integrity). These Byzantine agreement protocols use the common prefix property to ensure agreement and the chain quality one for validity. They also implicitly use the lower bound on the growth rate of blockchains described in [PSS16] (analysed in Section 3.1.1.2) to ensure termination. The second protocol is more complex than the first one in order to be robust against an adversary controlling half of the nodes, whereas the first one only tolerates a third of Byzantine nodes.

The authors define PoWs in an unusual way. First, they call difficulty Bitcoin's target, when Bitcoin's difficulty is actually inversely proportional to the target. Thus, increasing their difficulty makes it simpler to find blocks, which we deem needlessly confusing. Then, their explanation of how blocks are hashed is wrong. According to them, the hash of a block b is $H(\nu, G(s, c(b)))$ where ν is a nonce, s is the hash of $p(b)$ and H and G both correspond to SHA-256. In reality, G should return the concatenation of s and of the Merkle root of $c(b)$, and H should be SHA-256d.

This does not impact the validity of the analysis but we point it out as an example of the need for a clear, common and correct framework for academic studies of Bitcoin and, more generally, blockchain-based systems and protocols. On the other hand, the synchrony assumption impacts the usability of the model, especially over the global Internet.

3.1.1.2 Blockchains in partially synchronous networks

In Analysis of the Blockchain Protocol in Asynchronous Networks [PSS16], the authors present a formal model to study blockchain protocols, this time in partially synchronous networks. They prove upper and lower bounds on the growth rate of blockchains, a stronger property of consistency than that of Nakamoto [Nak08], and redefine chain quality with a formal definition of adversarial blocks.

The model assumes partial synchrony [DLS88] in a network made of a set of nodes whose composition may change with time: the adversary can choose which nodes to corrupt, limited only by an upper bound on the fraction of corrupted nodes in the network, and nodes may join and leave the network. Furthermore, nodes have identical computing powers; however, the adversary can coordinate the computations of the nodes she controls. The computing model is that nodes can query the random oracle to mine only once per time step but can make as many verification queries as they want. This seems a quite fragile assumption, as nodes could use the verification queries to actually mine by verifying, for a given message, whether its hash is equal to any value below the target in as many queries, thus virtually increasing their computing power.

The main difference between this model and ours is that it is application-agnostic: it focuses on the construction of the blockchain rather than what the application records in it. Thus, while our focus is on double-spending attacks, that are facilitated by attacks on the underlying protocols, theirs is specifically on malicious mining and its consequences on the composition of the blockchain.

3.1.2 Measures of the network

This section groups papers presenting the results of large scale measures on the Bitcoin network. They are complemented by trackers such as Blockchain.info [BC.I] and Blocktrail [BT], as well as projects such as Bitnodes [BN].

3.1.2.1 Measuring and improving information propagation

In Information Propagation in the Bitcoin Network [DW13], the authors describe the results of several measures performed on the Bitcoin network and possible protocol improvements. The main concern is the propagation of both blocks and transactions. Since transactions need to be propagated in order to reach miners and be confirmed, and blocks to keep the local blockchain replicas consistent, this is a matter of safety and liveness for Bitcoin.

The measures used a single router trying to establish 4000 outbound connections[1]. It stored every single block advertisement it received from the network, and assumed the first `inv` message announcing each block to correspond to the injection of the block in the network. The measure covered blocks 180 000 to 190 000, time stamped respectively on May 13$^{\text{th}}$, 2012 at 18:21:11 UTC and July 20$^{\text{th}}$, 2012 at 22:53:36 UTC. The router did not propagate information during the measure to avoid influencing the result.

The results were as follows: the median propagation time of a block was 6.5 s, the mean was 12.6 s, and the estimated probability density function (PDF) could be fitted by that of an exponential law with parameter 0.107[2].

The authors also measured the rate of forks and report a figure of 1.69 %. Using the assumptions that computing power is uniformly distributed in the network, that their measurements correctly represent block propagation in the network, and that the random skews in block time stamps are averaged out over their 10 000 blocks, they present a simple model for the occurrence of forks: they happen when a miner finds a block before receiving the current best. From that, they deduce that 1.80 % of the network's computing power is wasted because of propagation delays: the infamous 51 % attack would only need 49.1 % of the computing power to succeed.

This corresponds to the phenomenon called the *Blockchain Anomaly* [NG16] an attacker able to use networking power to slow down information propagation can lower the computing power needed to rewrite an arbitrary length of the blockchain.

Then, the paper describes a few ways the protocol could be improved to reduce the propagation delays. First, transaction could be directly advertised, without going through the 3-way handshake as it represents the main part of the propagation time for small messages such as transactions. This method could potentially work without creating any vulnerability, but it would increase Bitcoin's network resource consumption as any router with 8 neighbours would require the transaction to be sent at least 8 times, distributed between to and from him depending on the relative reception times of the transaction in his neighbourhood, compared to possibly only once and 8 inventory (whose size is significantly smaller than that of transactions since it is strictly smaller than each input): determining whether the trade off is acceptable is not trivial despite the fact that it could improve Bitcoin's liveness by reducing the time between the emission of a transaction and the instant at which miners start confirming it.

Then, the propagation of blocks could be improved in two ways by routers: first, each router could propagate blocks as soon as they pass the context-independent validity checks (see Appendix A.4.4) rather than waiting for them to pass the complete validity check; then, routers could relay block advertisements as soon as they receive one. Since the most difficult part in finding a block is completing the PoW, whose verification is included in the context-independent validity checks, the former would

[1] The figure is not mentioned in the description of this first experiment, we assume it to be equal to that of the second one.

[2] This value, not included in the paper, was provided by the first author by e-mail.

not expose routers to DoS attacks; malicious peers aiming at maximising the effect of their misbehaviour would use their computing power to find valid blocks rather than invalid ones. Adapting the peers' source code to include that modification would not be difficult. Moreover, since SPV mining is used by several pool [BW.MP], one could argue that it would improve the current situation. This would indeed improve the delays in the network, but at the cost of an increased resource consumption.

The second proposal can be analysed the same way except that it provides a vector for DoS attacks. Indeed, advertisements would flood the network and they are free. Though well-formed transactions require more work from the receiver to perform the validity check, nodes drop transactions whose fee is insufficient to get confirmed instead of propagating them.

Finally, a third solution is examined: using highly connected routers to drastically reduce the diameter of the underlying graph. Though efficient, this is not feasible in the long run: the infrastructure required to handle the bursty network load is expensive and threatens Bitcoin's decentralisation.

The third aspect of the paper describes the eclipse phenomenon: to avoid consuming network resources, routers do not propagate what they consider invalid information: locally invalid or non-standard transactions (*e.g.* double-spending attempts) and concurrent blocks during forks. Whenever a peer detects a conflict, it picks a side and considers the other one non-existent until proven wrong. This saves network resources and prevents some flooding attacks: no attacker can perform a DoS attack at the network scale by propagating several transactions consuming the same input and paying only once the fee. On the other hand, it helps perform double-spending attacks since only the routers neighbouring the cut between the two subgraphs defined by which of the two versions of the conflict is accepted actually know that the cut exists. In most common scenarios, propagating everything independently of its contextual validity (*e.g.* both concurrent blocks during a fork) as long as it is context-independently valid would alleviate this issue, and vendors could make sure not to provide their goods in exchange of a transaction that is not deeply confirmed. However, this would not provide any protection in case of a network split, quite the opposite: it would give a false feeling of safety.

All in all, this paper provides an insight in what the Bitcoin network was in 2012. Since then, the protocol has slightly evolved, with among other a modification in the advertisement of blocks. It highlights ways to improve Bitcoin's flooding mechanism but most of them tend to expose the network to flooding attacks or, at least, to make them easier to perform and more efficient: it improves Bitcoin's liveness and safety properties when all peers behave properly but may harm them in the presence of malicious ones.

3.1.2.2 Bitcoin's network topology

In Discovering Bitcoin's public topology and influential nodes [MLPG+15], the authors present the results of two related experiments they performed on the Bitcoin network through the infrastructure they call CoinScope. First, they map the struc-

ture of the network's graph; then, they search for the routers that were communicating with the highest shares of the effective computing power.

Mapping Bitcoin's graph relies heavily on the address propagation mechanism described in Appendix C.2. In short, a router can ask its neighbours for a list of addresses through `getaddr` messages. The neighbours answer with `addr` messages by sampling a limited subset of their address database. CoinScope then infers whether a connection is established between two routers depending on the time stamps associated with each address.

With this process, between 4000 and 7000 routers were probed over 18 days. The results show that most reachable routers had between 8 and 12 neighbours, some having up to slightly less than 1000. Most of these outliers are identified as belonging to mining pools (mostly the Bitcoin Affiliate Network [BAF]) or wallet services. The authors conclude that the Bitcoin graph significantly differs from a truly random one.

This first experiment requires a few comments. First, the number of reachable routers is significantly higher than that measured by Decker *et al.* [DW13] (analysed in Section 3.1.2.1), with a mean in the order of 5000 routers instead of 3048. The former is consistent with the usual estimations provided by Bitnodes [BN] while the experiment we describe in Section 3.2 is closer to the latter. We briefly discuss possible origins of this discrepancy in Section 3.2.3. Second, the Bitcoin Affiliate Network, which was linked to 29 % of the highest degree routers in the snapshot shown in the paper, has since then disappeared from the Bitcoin horizon: Blocktrail [BT] reports that the last block they found was on Wednesday, December 2$^{\text{nd}}$, 2015, at 4:24:42 GMT.

However, the main issue resides in their comparison with a truly random graph: they provide no formal definition of such a concept. Indeed, there are many different models for random graphs [BR05; BA99; ER60; Gil59] which exhibit significantly different features. The closest model to what we expect from the Bitcoin graph is the undirected growing 8-out [BR05] one, where each vertex successively joins the network and connects to 8 vertices. However, this does not take into account the very well connected vertices that they have found, and the claim that their influence is minimal is dubious: if 50 vertices have an average degree of 200 in a network also comprising 5000 vertices of degree 8, then those 50 well-connected vertices hold 20 % of the graph's edges. Though the figures provided in this toy example are arbitrary, they comply with the data provided of 48 nodes with degree ranging from 90 to 708; thus, they should either be part of the argument against the graph's *true randomness* (which still lacks a formal definition) or be taken into account in the random graph model.

Another potentially important pitfall of their approach is their assumption that most nodes use the same unintuitive time stamp scheme as Core for addresses (see Appendix C.2). Given that it depends on mostly undocumented behaviour, it seems reasonable to assume that some clients handle it differently. Additionally, Core v0.13.0 only tolerates one `getaddr` request per inbound connection, which makes the experiment much more complex to perform.

The second experiment, finding the influential routers, rely on sending different transactions all consuming the same inputs to different parts of the network, and identifying which of those get included in the blockchain. Indeed, under a uniformly distributed computing power assumption, for each set of conflicting transactions, the winner is chosen uniformly at random. The actual distribution of winning transactions based on where they have been sent can provide an insight as to the actual distribution of the computing power in the network.

The results of this experiment are that less than 2 % of the routers were strongly linked with almost 75 % of the total computing power of the network, and many of those routers are associable with specific mining pools or, at least, Bitcoin addresses to which their coinbase transactions send their output.

Once again, the churn in mining pools is highlighted by the presence of GHash.io among the ranks of the most powerful pools at the time; it has since then greatly fallen behind and has been associated with barely 9 blocks between heights 429 000 and 430 000 (0.9 %) [BT]. However, this experiment relies on a bug described by Gervais *et al.* [GRKC15] (analysed in Section 3.1.3.2) that Core v0.12.1 fixed.

This experiment shows that the assumption of uniformly distributed computing power is unrealistic. However, given that those routers do not seem to have any other distinctive feature, one could argue that dynamic graph models can still use the assumption and consider that it represents the probability that the influential nodes are at a given location at a given time rather than that each node has the same amount of computing power.

3.1.3 Information propagation

This section describes two papers focusing on Bitcoin's flooding protocol, pointing out its flaws and possible solutions. Given the lack of formal specification, most papers focus on bigger overall modifications of Bitcoin to improve its characteristics; Section 3.1.6 describes some of them.

3.1.3.1 Incentive for information propagation

In On Bitcoin and Red Balloons [BDOZ11], the authors describe a possible incentive for information propagation, to account for the fact that it consumes resources to broadcast transactions at no gain for any router that is not concerned by it. It is even quite the opposite: if miners have more unconfirmed transactions than they can include in a single block, they can choose the ones with the highest fees to maximise their own profit, which leads to a competition between wallets to find the lowest transaction fee that will get their transaction confirmed at the minimal cost for them. In this scenario, not propagating competing transactions is a way to increase one's chances to get one's transactions confirmed.

One way to fix this issue is to make it interesting for routers to propagate all transactions. Just as miners are rewarded for each block they find, routers would be rewarded for each transaction they help propagating to the miner that eventually

manages to confirm them. Because of the openness of the Bitcoin network, the authors claim that their scheme is Sybil-proof: it is not beneficial to generate a large number of identities not backed by actual networking resources.

The proposition is that whenever a router relays a transaction, it can add its identity to a chain of signatures included in the transaction. The chain has a limited height and, when the transaction is confirmed in a block, the system rewards all the accounts that signed it. The height and the reward are parameters of the scheme; the actual scheme uses two sets of parameters in parallel and the authors show that, in their model, it is more profitable for routers not to duplicate themselves in the signature chain because it increases their chances of seeing the transactions they propagate included in a block.

This could improve the propagation time of transactions in the network and, thus, help detect double-spending attempts. However, it suffers from several drawbacks. First, its model is quite peculiar: instead of the usual random (or regular) graph, it is based on a forest of complete d-ary trees. It also considers that the protocol is divided in two phases: first, all nodes propagate as many transactions as they want to all their neighbours and then all nodes mine until one finds a block. Finally, the computing power is uniformly distributed among the peers, which are all nodes.

The authors provide experimental validation neither for their model nor for their scheme; they claim that rumour spreading is harder in a tree than in a graph as each node has total control over the data flow to its children. However, combining this structure with the assumption of uniformly distributed computing power gives theoretical results of possibly limited applicability: this may instead lead to a competition to get as close as possible to the influential nodes discovered by Miller *et al.* [MLPG+15]. The effect of this reorganisation on the resilience of the network would need additional investigation. Assuming that miners start to try and confirm transactions after all miners have received them is reasonable in that transactions propagate quickly compared to the time needed to find a block.

Another adverse effect of this solution is shared by most incentive schemes: if incentives are distributed for a given action, it can be expected that said action will not be performed without retribution any more. Thus, routers receiving a transaction after the signature chain has reached capacity will probably not even try to propagate it: the capacity of the signature chain must be chosen accordingly. This can also be leveraged to increase the success probability of a double-spending attack: Alice can establish a direct connection to Bob, the node controlled by a vendor Frank selling her user Gina some goods[3], without Bob knowing the association between Alice and Gina. Then, sending Bob the transaction with a signature chain already almost at capacity lets Frank believe that it is propagating; Alice can in parallel broadcast a conflicting transaction with an empty signature chain. To Gina, in the worst case scenario, the legitimate transaction gets confirmed (*e.g.* Bob successfully

[3]Many IPv4 addresses can be associated with geographical addresses with a sufficiently good precision for this not to seem unrealistic.

mines it) and she still gets most of the propagation incentive back; more likely, the illegitimate transaction is successful, along with the attack. In short, the Sybil-proofness of the scheme only applies to intermediate routers propagating a valid transaction and leaves out some potentially harmful cases.

Finally, there are two more issues from the implementation point of view. First, it requires the emitter of each transaction to have at least 15 neighbours, while the current lower limit for the Core is 8: this number would need to be raised. According to Miller *et al.* [MLPG+15], though, this may not be an issue if most routers currently only have a tenth of their maximum number of connections. On the other hand, how the chain of signatures should be implemented is unclear. The authors suggest replicating the field, in a transaction's body, used to give the fee to the miner confirming the transaction. As shown by Appendix A.3, there is no such field. A supplementary output, with the right scriptPubKey, could be still be a possible idea. However, there are many issues to solve:

1. To preserve privacy, it should be difficult to associate a router with the wallet of its controlling user. A signature chain would make each router advertise a public key in the chain that the wallet would then use, associating the two;

2. the signature chain cannot be covered by the emitter's signature and would need to enforce its own tamper-proofness; though S-BGP [KLS00] manages something similar to what would be needed here, its approach is probably not scalable enough and regular routers could not process a large number of transactions per second, which would decrease Bitcoin's throughput;

3. claiming the funds would be tricky as well: currently, each output can only be claimed once. Here, the protocol would either require from each block to include several coinbase-like transactions rewarding the appropriate routers (along with the associated validity checks performed by each node) or from that special output to be claimable once by each signature in its chain.

In conclusion, though based on a good idea, the authors of [BDOZ11] do not solve the issue they tackle because of both theoretical and implementation-related unsolved problems. Their proposal has no effect on Bitcoin's safety because it restricts itself to the diffusion of loose transactions in the network, and because of its drawbacks its effect on Bitcoin's liveness can only be negative: non-conflictual transactions might not reach a single miner if the incentive to propagate it is not large enough and nodes, rationally deciding to only propagate rewarding transactions, could drop it. If those issues were to be fixed, it could potentially improve Bitcoin's liveness by decreasing the delay between the emission of a transaction and its reception by the miner that will succeed in confirming it. For this effect to be significant would however require that the propagation delay of a transaction be non-negligible compared to the time needed to find a block, which goes against one of the assumptions of the model, or that a significant portion of transactions that should be propagated be dropped without this incentive.

3.1.3.2 Tampering with information propagation

In Tampering with the Delivery of Blocks and Transactions in Bitcoin [GRKC15], the authors focus on a special kind of DoS attack to which the Bitcoin protocol is vulnerable, and how it can be used to increase one's mining revenue and chances of success for double-spending attacks. The vulnerable mechanism is the 3-way data exchange and its time-out detection.

Indeed, as the authors point out, when Bob requests a transaction from Alice in response to her advertisement, he waits for 2 minutes before requesting it from another neighbour; thus, Alice can easily withhold the item for that long. However, cascading this attack by sending several advertisements for the withheld transaction does not work any more since pull request 7079, by Gregory Maxwell, was merged into the source code of the reference client [Core]: Bob now filters the queue he uses to know which neighbour to ask for a given transaction so that a single router can only appear once per transaction. This would make the double-spending detection prevention attack, consisting in withholding the illegitimate transaction from Bob until it gets confirmed, unusable as he would learn about it right after the time out unless Alice managed to perform the attack from several routers in Bob's neighbourhood. However, since Core routers only propagates one transaction in case of conflict, conflict detection is already a difficult task without this attack. According to Bitnodes [BN], on Wednesday, August 31st, 2016 at 19:03:21 GMT, Bitcoin XT, a client that propagates all transactions involved in a conflict to let other nodes detect the conflict, was run by less than 100 nodes (less than 2 % of the 5272 nodes seen in the network by the tracker at that time).

The paper is also slightly outdated as regards block propagation for the same reason: some of the solutions it recommends have been implemented. The attack is quite similar: to prevent Carol from receiving a block, Bob can send her the corresponding advertisement and not follow up with the actual block. Since nodes do not register block advertisements for blocks they are waiting to receive, if Bob manages to be the first to send the advertisement, there is a high chance that all of Carol's other neighbours will advertise the block before Bob's transmission (or lack thereof) times out and Carol will need to wait for the next block to be propagated (or another neighbour to connect to her) to be able to receive the missed block. Thus, the attack is even more powerful for blocks than for transactions.

However, the situation has changed in two ways. First, Core v0.12.1 uses a time out for block reception of 10 minutes plus 5 per other neighbour sending a block whose header has already been validated instead of the previous 20 minutes. In a regular setting, where nodes need only download one block at a time, that amounts to half of the previous value and Carol would disconnect from Bob as soon as the transmission times out, not allowing him to perform it several times in a row without using several colluding routers. Then, the whole mechanism for propagating blocks has changed: the first message of the three-way handshake contains the header of the block instead of a simple advertisement. This way, nodes can only request blocks that seem valid, and Bob would need to completely eclipse Carol from all of her

honest neighbours to prevent her from receiving the block; among others, he would need to get her to make all of her 8 outbound connections to routers colluding with him.

Given these protocol updates, the attacks reported by the paper are much more difficult to perform, and thus the mining and double-spending advantages are reduced though not completely voided. The mining advantage is even more reduced by SPV mining when performed by getting block hashes directly from other pools' websites, even though this is considered bad practice. Out of the eight recommended counter-measures, four have been implemented. The remaining four are to use dynamic time-outs, adapted to each router's connection for a better detection of stalling and withholding, to choose randomly the recipients when sending transactions requests (and linearly increasing their number for withheld transactions) rather than using a queue, to use several nodes and to consider non-responding a bad behaviour, with an associated penalty. All but the third one are implementable and could work for each individual node implementing them; the third one is a matter of user behaviour rather than implementation.

In conclusion, the recommendations from this paper helped improve Bitcoin's liveness by patching a DoS vulnerability and its safety by reducing an attacker's capability to use its networking power to increase its relative computing power by decreasing the effective computing power of the network. Performing the same measures and experiments on today's network would be interesting in order to measure the real impact that the deployed counter-measures have had, taking into account that not all nodes have implemented them.

3.1.4 Malicious mining

This sections groups two papers that studied mining strategies to misuse the blockchain in order to increase one's expected profits or mount double-spending attacks, along with their success probabilities. Many more article have been published on the topic of mining, either to describe more adversarial behaviours [ES14b; JLGVM14; Eya15], or to study diverse topics such as economics or ecology [KDF13; OM14].

3.1.4.1 Nakamoto's white paper

In Bitcoin: a peer-to-peer electronic cash system [Nak08], Nakamoto gives a proof of concept for Bitcoin, describing transactions, blocks, the blockchain, and the risk of malicious forks in order to commit double-spending attacks.

His model implicitly assumes two competing entities: the honest nodes and the Byzantine ones. Each entity mines synchronously: when a node finds a block, all the other of the same entity instantly start working on it. The goal of the Byzantine nodes is to produce a blockchain longer than that of the honest ones.

Nakamoto uses this model to compute suitable deep-confirmation thresholds depending on the power of the adversary: this led to Bitcoin's current value of 6. In Nakamoto's model, an adversary controlling 10 % of the computing power has a

success probability less than 0.1 %; that of one with 30 % of the computing power is greater than 17.7 %.

Given the strong assumption of synchrony and the very weak adversary that barely deviates from the (implicit) protocol, and given the current computing powers of mining pools, the biggest one consistently controlling more than 15 % of the computing power [BC.I; BT], the trust in Bitcoin's current deep-confirmation threshold may be overly confident. Nonetheless, this paper has provided the first model of attacks on Bitcoin along with the system.

3.1.4.2 Selfish mining

In Majority Is Not Enough: Bitcoin Mining Is Vulnerable [ES14a], the authors describe a much more efficient mining strategy that can increase the revenue of a pool and requires much less than 50 % of the network's total computing power.

The model makes the following assumptions: the system comprises a fixed number of miners. Some of them, controlling a fraction μ of the total computing power, collude to form a selfish mining pool. The propagation time of blocks is considered negligible in front of the time it takes to find them: there are no accidental forks caused by the honest miners and when one is triggered by the selfish miners, a fraction γ of the honest computing power chooses to mine on top of the selfish branch. It also assumes the absence of target adjustment.

We have described the selfish mining strategy in Section 2.3.5. It consists, for the selfish pool, in not releasing blocks when they are found but only when, if kept secret any longer, they would with high probability be pruned out of the blockchain. Thus, the action taken by the selfish pool when a miner found a block depends on the length of the public blockchain, on the number of blocks it has managed to find on its own and has not released yet, and on whether or not the successful miner is a member of the pool.

The authors prove that the efficiency of this strategy depends on γ, the fraction of the honest computing power that sides with the selfish pool when it triggers a fork. The difference between the profits from the selfish and "normal" mining strategies increases with μ; the break-even point corresponds to the μ such that the two strategies are equally profitable. When $\gamma = 0$, the break-even point corresponds to $\mu = 1/3$. As γ grows, the selfish pool reaches the break-even point for lower values of μ, down to $\mu = 0$ for $\gamma = 1$.

The authors further present a simple modification to the honest strategy, consisting in randomly choosing the block on which to mine in case of fork with branches of equal weight instead of Bitcoin's first-arrived policy. This modification ensures an expected γ of 0.5, while the first-arrived policy gives the upper-hand to pools with broadcast advantages. Through this, the threshold at which the selfish strategy becomes better than the honest ones is ensured to be 0.25.

Finally, given that the selfish pool has higher revenues than the other ones, it can be expected that rational miners will join it to increase their own revenues. Given that this also increases the revenues of the miners that were already in the pool, they

have an incentive to let anyone in; thus, as soon as a pool gathers enough power to use the selfish strategy, it can be expected to grow, reach the 50 % threshold and overtake the whole system.

The validity of the model derives from its optimistic assumptions: accidental forks help the selfish pool elongate its secret blockchain at a faster pace than the honest miners do the public one. The absence of target adjustment simplifies the strategy but does not significantly changes the conclusions: the worst case scenario for the selfish pool is to lose a fork every 2016 blocks.

However, the selfish mining strategy does not in itself threaten Bitcoin's fundamental properties as defined in Section 2.2. Indeed, it decreases the *fairness* of mining but still only produces well-formed blocks, and even if the selfish pool selectively denies service to some conflict-free transactions, its expected fraction of the linearised blockchain remains strictly less than 100 % and they would be confirmed in the blocks found by the remaining honest miners. Despite this, it does increase the probability of forks, which in turn increases the time needed for transactions to get deeply-confirmed, and it increases the risk of 51 % attacks that our model does not formally capture.

3.1.5 Double-spending attacks

This section groups two papers that set double-spending attacks as their main field of study. Other papers tackling the issue tend to either analyse Bitcoin or blockchains in a more general setting [Ros14] or suggest broad protocol modifications [KJGK+16; DSW16] to prevent double-spending attacks.

3.1.5.1 The insecurity of fast payments

In Double-Spending Fast Payments in Bitcoin [KAC12], the authors describe the simplest scenario for a successful double-spending attack, where vendors do not wait for transactions to be confirmed before providing their goods. They show the feasibility of such attacks and the possibility for attackers to succeed and remain undetected and describe three countermeasures to prevent these attacks.

In order to emphasize the feasibility of the attack, they consider a very weak adversary, who controls between two and six routers and no miner. Out of those routers, one maintains a single connection with the vendor while the others maintained between 125 and 400 connections and made sure not to have one established with the vendor. Such an adversary manages to perform double-spending attacks with overwhelming probability against vendors that accept a transaction as soon as they receive it rather than waiting for at least one confirmation.

The three countermeasures they describe to mitigate those attacks are as follow: vendors should wait a few seconds after they have received a transaction to confirm that no conflicting one is being propagated; they should use several routers, located at different places of the network, and verify that even in the union of their views, the transactions funding them are not conflicting; finally, all routers should propagate

all well-formed transactions, even the conflicting ones. The first one alone would not work: as soon as the vendor receives the transaction, she forwards it to all her neighbours who would detect the double-spending attempt but not warn the vendor. The other two are usual recommendations, discussed as well in [GRKC15] (analysed in section 3.1.3.2).

Three assertions throughout the paper seem surprising to us. First, they assume a network of 60 000 nodes based on an estimate provided by Bitcoin Wiki [BW] using the size of the address manager of an arbitrary node. This overestimates the number of devices running a node for at least two reasons: a single machine can have several addresses, and this includes SPV nodes who do not propagate information. Moreover, the value is tenfold that reported by the authors of [DW13] barely one year later; however, the estimate is not used to analyse the results of their experiments. Then, they use the fact that IP addresses are public because they can be sent funds; Core has since then disabled this feature, deemed too insecure. It makes it more difficult for an attacker to find the node of a double-spending target but, given the mapping between location and IPv4 addresses, it is still feasible unless vendors use techniques such as VPNs to have routers at addresses that do not correspond to their physical locations. Third, they use the anonymity of Bitcoin and the unlinkability of addresses to state that attackers can perform double-spending attacks in total impunity; studies have shown the limits of both properties [BKP14; AKRSC13].

This paper was later extended in [KARGC15] but the conclusions remain: accepting transactions with a confirmation level of 0 is highly insecure.

3.1.5.2 Securing fast payments

In Have a Snack, Pay with Bitcoins [BDEWW13], the authors present a solution to the double-spending issue for attackers that do not try to fork the blockchain. They present an experimental validation of their solution and demonstrate its feasibility with a prototype of vending machine implementing it.

Their model is simple: the attacker cannot map the network to find the neighbours of a specific router and cannot disturb communications, but she can connect to an arbitrary number of routers and broadcast at will. She does not mine. On the other hand, the vendor's router does not accept inbound connections and does not propagate transactions funding keys managed by its wallet to avoid the self-eclipsing phenomenon pointed out by Decker *et al.* [DW13].

Using this model, they evaluate the success probability of double-spending attacks similar to those described by Karame *et al.* [KAC12]. The attacker uses two routers which release pairs of conflicting transactions at random places in the network. The vendor maintains 1024 connections on average and the number of simulations is increased by selecting, for each double-spending attempt, a random subset of those as the neighbours for a simulation. Each subset comprised at most 100 neighbours.

In these conditions, they report that a vendor maintaining 100 connections will learn of double-spending attempts with overwhelming probability (99.23 %); that

99 % of double-spending attempts were detected before receiving 37 announcements for a transaction or at most 6.29 s after receiving the first announcement. Should vendors wait for both these conditions to hold, they report that double-spending attempts would only succeed 0.088 % of the time. They demonstrated the feasibility of such a strategy by implementing it in a snack vending machine.

Though this works in their scenario, one should not forget that this double-spending scenario is really simple and more elaborate ones include forking the network. Moreover, Miller *et al.* have shown that it is feasible to map the network [MLPG+15]. Short of being able to let the merchant eclipse herself, an attacker could send the illicit transaction to the influential nodes pointed out in the same paper and let the transaction funding the vendor propagate normally in the network; the chances that the vendor discovers the attack can be further decreased by using several routers to broadcast the licit transaction, since the illicit one has a much greater chance of being confirmed.

Another issue is the consumption of network resources: if every seller (*e.g.* store or vending machine) runs such a modified peer, establishing 100 outbound connections and accepting no inbound ones, the network may run out of open connection slots. Indeed, a network of n regular routers, all establishing 8 outbound connections and maintaining at most 125 parallel connections can tolerate up to $1.17n$ such modified peers before complete exhaustion of the open slots, preventing new peers from joining it. This number further decreases when taking into account the other nodes that do not accept inbound connections because *e.g.* of firewalls. Moreover, the size of the network is already deemed concerning [Caw14]: despite being an interesting first step towards a more secure Bitcoin, this proposition does not solve everything.

3.1.6 Protocol

This last section groups three papers that deal with specific aspects of the protocols used by Bitcoin to generate and propagate data such as blocks. Many more focus *e.g.* on different ways to modify the mining process to make it fairer [PS16], discourage the formation of pools [ES14c] or more resilient against selfish mining [Hei14].

3.1.6.1 Safety analysis

In Safety Analysis of Bitcoin Improvement Proposals [ALLS16], we have introduced a simple model to formally define the concept of double-spending attacks, and we have analysed the safety, or lack thereof, of three recent works: Bitcoin-NG [EGSVR16], PeerCensus/Discoin [DSW16], and ByzCoin [KJGK+16].

The model is but a simplification of the one we have described in Section 2.2. Indeed, a number of concepts such as coinbase transactions are not as fully described as they should be to completely model the actual system. Nonetheless, it laid the foundations of our model for Bitcoin in an adversarial network.

The main contribution that is not directly continued by this work is the second one, the analysis of Bitcoin-NG, Discoin, and ByzCoin as regards their relying on miners to improve Bitcoin's way of processing transactions. Briefly, the three protocols all suggest using a supervising group $\mathcal{E}_\ell$ to validate transactions on the fly instead of in slowly-generated blocks. Every time a wallet creates a transaction, it sends it to $\mathcal{E}_\ell$ rather than to all the miners; if validated, the transaction is instantly confirmed and definitively set in the history of the system. Such a system provides much better safety and liveness properties as deep-confirmation is not required any more. The supervising group consists of the last ℓ successful miners, where ℓ is the main conceptual difference between the three protocols: Bitcoin-NG sets it to 1, Discoin to ∞ (*i.e.* all the successful miners) and ByzCoin to w, treated as a security parameter. Thus, whenever a block is found (and, for Discoin and ByzCoin, accepted by $\mathcal{E}_\ell$), the composition of $\mathcal{E}_\ell$ changes.

The issue with those three systems is that they are all insecure. Indeed, the probability that the blockchain only contains blocks found by non-malicious miners is close to zero simply because of its length, and Bitcoin-NG gives much greater power to miners than Bitcoin does as they get the ability to validate *future* transactions instead of past ones when they successfully find blocks. Discoin relies on the fact that, in its permanent regime, the proportion of malicious entities in $\mathcal{E}_\infty$ is the same as the proportion of malicious computing power, but this does not take into account the fact that in most trajectories from the initialisation of the system to its permanent regime, $\mathcal{E}_\infty$ becomes polluted (*i.e.* more than a third of its entities are malicious, which is a well-known bound for the impossibility to reach consensus). Even worse, as soon as it is polluted, its malicious entities may use their presence to veto blocks found by non-malicious miners and only accept their own, further increasing their power over the system. Similarly, ByzCoin reaches polluted states but the length of the window can be adjusted to increase the probability of safe executions depending on the fraction of malicious miners.

Many more things can be said about those three protocols: for example, they also present scalability issues, and Bitcoin-NG implements a denunciation scheme to condemn malicious $\mathcal{E}_1$'s. However, the point remains that they do not improve Bitcoin's fundamental properties because of shortcomings in the handling of malicious miners.

3.1.6.2 The GHOST rule

In Secure High-Rate Transaction Processing in Bitcoin [SZ15], the authors present a modification of Bitcoin's fork resolution method to allow the system to increase the rate at which miners find blocks without jeopardising the safety of the blockchain.

They model the network as a directed and weighted graph. There are communication delays corresponding to the weights of the edges, miners do not necessarily have the same computing power and the attacker follows Nakamoto's misbehaving protocol [Nak08]: she aims at creating a secret branch of the blockchain to release it at a later point in time and win the fork. Her expected block generation rate

is constant and she does not accidentally fork her secret chain. They do not take target adjustment into account.

The protocol they describe, called GHOST, changes the blockchain linearisation method by redefining our simplified pseudo-confirmation level: instead of taking the longest path rooted by a block, they propose to compute the size of the subtree it roots. Formally, for a block b, the simplified pseudo-confirmation level becomes

$$L'_b = |\{b' | \exists k \in \mathbb{N}, \exists b_0, ...b_k \in \mathcal{B}, b_0 = b \wedge \forall i \in [[1, k]], p(b_i) = b_{i-1} \wedge b_k = b'\}|$$

instead of Bitcoin's

$$L'_b = \max\{k + 1 | \exists k \in \mathbb{N}, \exists b_0, ...b_k \in \mathcal{B}, b_0 = b \wedge \forall i \in [[1, k]], p(b_i) = b_{i-1}\}.$$

This modified pseudo-confirmation level is then used to prune branches the same way Bitcoin does. When forks only involve two conflicting branches, the two rules are equivalent. As soon as multiple forks occur, they prove that their rule is safer. Indeed, if 60 % of the total computing power work on a branch A and the other 40 % on a branch B, then the expected winner of the fork is branch A; however, if A is forked before B is pruned out, and half of the computing power dedicated to it starts mining on a branch C, then B becomes the expected winner of the fork even though the subtree containing A and C at the beginning of the fork between A and B has received more computing power. With GHOST, B is still pruned out because the combined weight of A and C is greater than that of B.

This makes a non-negligible difference with their adversary model: with GHOST, the effective computing power of the network remains equal to the total computing power instead of being divided by 2 in the worst-case scenario. Since the success probability of the attack depends on the ratio between the computing power of the adversary and the effective computing power of the network, it is clear that the system is safer with GHOST.

This increased safety comes with the added benefit of an increased scalability: since accidental forks do not threaten the safety of the network, the generation rate and size of blocks can be increased with jeopardising the system. On the other hand, increasing those parameters too much still decreases the efficiency of the network as all nodes must receive all blocks to compute the pseudo-confirmation levels, even though the blockchains is still eventually linearised.

Thus, the weakest point of this solution is that it offers scalability at a huge cost in efficiency: achieving 9.09 transactions per second instead of Bitcoin's current 3 decreases the efficiency, computed as the growth rate of the main chain divided by the expected block generation rate, from approximately 1 (there are currently very few accidental forks, see *e.g.* Sections 3.1.2.1 and 3.2.2) to 0.2. Given how Bitcoin is already criticised for its wasteful PoW mechanism [OM14], this may not be an acceptable trade-off. Additionally, the adversary model is very limited: the impact of more complex mining strategies should be evaluated as well.

As regards Bitcoin's fundamental properties, its impact seem to reside mostly in the range of viable choices for the security parameters. Thus, though it may improve

Bitcoin's liveness and safety by decreasing the time needed to resolve repeated forks (because the pseudo-confirmation level of exactly one of the blocks responsible for the fork increases whenever a miner finds a block, instead of "at most one"), it does not fundamentally change them.

3.1.6.3 Proof of Stake

In Cryptocurrencies without Proof of Work [BGM14], the authors present a way to generate blocks at a regulated pace that is much less resource consuming than Bitcoin's PoW: the proof of stake (PoS). Intuitively, while a PoW scheme awards blocks to miners proportionally to the work they invest in the system, a PoS one awards blocks to *stake holders* proportionally to the stakes of the system they own.

A simplistic PoS protocol would use a block as the seed of a random number generator to pick uniformly at random a satoshi, and its current owner would be the only peer allowed to generate a new block, the *lucky stakeholder* (as opposed to the successful miner of PoW schemes). For many reasons, this would be highly impractical: it would with high probability quickly choose an unspendable coin (*e.g.* one whose private key was lost), and the system would stale indefinitely.

Thus, the Chains of Activity protocol developed by Bentov *et al.* is more involved. First, a single seed is used along with a counter to elect several consecutive lucky stakeholders, so that one can be skipped if she takes too long to produce a block. Similarly, several consecutive blocks are combined to generate a single seed to prevent attackers from crafting a seed giving them back the right to generate new blocks: blocks are packed in groups of equal size, and an entire group is used to generate the seed used to determine the lucky stakeholders of a following group. Additionally, groups are interleaved: the k-th group generates the seed of the $k + 2$-th group. Finally, a punishment scheme is used to sanction the malicious lucky stakeholders that generate pairs of conflicting blocks.

The system relies on the following security parameters: the size of block groups, the minimum amount of time between two consecutive blocks, the function used to derive a seed from a group of blocks, the punishment for conflicting blocks, the minimum amount of coins to engage as a PoS, and the time these coins are frozen to prevent double-spending attacks.

The authors point out a number of attacks on the system: lucky stakeholders could collude to secretly fork the blockchain in order to mount double-spending attacks, anyone could try and bribe the stakeholders into letting one perform a double-spending attack, or lucky stakeholders could craft blocks in such a way that the system would pick their satoshis again. The first two attacks are also possible with PoWs but much more unlikely since it would consume resources to perform them whereas they are costless in PoS schemes.

They also suggest countermeasures: the interleaving of groups increases the minimum size a collusion must reach before threatening the system. Similarly to any currency, if too many abuses are discovered, the value of the currency drops; however, they argue that since the currency itself is used to elect lucky stakeholders, an

attack would be extremely costly and leave the attacker with an enormous amount of worthless coins. This is slightly stronger than the argument used for PoWs schemes as the malicious miners still have their mining equipment, which can be repurposed. Finally, they suggest to use checkpoints, much like Bitcoin used to: known blocks at given heights of the blockchain which are unequivocally and definitively agreed upon; forks can only take place after the last checkpoint.

However, a few issues are left: checkpoints are hard to implement in a secure and completely decentralised way; bootstraping the system (*i.e.* the initial money distribution) is not feasible in a fair way with a pure PoS system as any solution would necessarily favour the early adopters; forks cannot be solved because stakeholders have no incentive to focus on extending a single branch since the process is costless; finally, it conflicts with *cold storage*, Bitcoin's security recommendation to keep most of one's coins on an account whose private key is kept encrypted and offline: signing a block with such a key would be impossible to automate.

This last issue can easily be solved, though, by including a challenge such that a signature with a key K_1 would be required to spend the funds but another signature with a key K_2 would suffice to prove ownership of the funds but not to spend them. That way, K_1 may be kept encrypted while keeping K_2 available to the Bitcoin client, ready to sign blocks awarded to the account.

Finally, the PoS scheme has a lot of advantages, if only from the ecological point of view, but some work is still required to make it usable and secure enough to be used for a process as critical as the generation of blocks.

3.2 Measuring Bitcoin's network

Two parameters are of critical importance to Bitcoin's safety: the total hashing power, preventing adversaries from taking over the blockchain, and the block propagation time, related to the occurrence of non-malicious forks. An aggregated measure of the two can be obtained as the network target, computed by all nodes and indicated in the header of all blocks. Given that mining pools hide their power from the network (to avoid appearing capable of 51 % attacks and being the target of DoS attacks) and that the total hashing power used to mine is dynamically distributed over several altcoins, measuring an exact total computing power would be quite difficult.

However, Decker *et al.* [DW13; CDEG+16] have measured the propagation time of blocks in the Bitcoin network. As describes Section 3.1.2.1, we consider that part of the data is missing from the results. Since, additionally, Bitcoin Core's networking protocol has changed since their experiments, we repeated it, with a few modifications.

3.2.1 Method

A modified version of Bitcoin Core was run on a machine (called Parasolier in the following) equipped with a 2.80 GHz Intel Xeon CPU (model E5-1603 v3), 8 GiB of

RAM and 8 GiB of swap memory, and a 1 GiB s^{-1} Intel Ethernet Connection I217-LM. The base source code was that of Bitcoin Core v0.13.0rc1 [Core]. Three types of modifications were made: first, most messages uploading data were blocked right before being serialised, letting only control and `get*` messages pass through; then, inventories, blocks and transactions were all recorded by an additional thread. Inventories were recorded along with the time stamp of when the message was received by the client (in microseconds), blocks with the list of neighbours at reception time and transactions as decoded strings. The instant at which each block was logged by the recording thread was recorded as well, as an approximation of its time of reception. Finally, the client was reparametrised to establish more connections, which required to change all `select` structures to use `poll` instead because the former is limited to 1024 sockets, and used multiple threads to establish connections instead of just one. Each of those follows a procedure similar to that of Core (described in Appendix C.1), expect that it skips all sleep periods and tries all reachable addresses picked by the address manager.

Thus, our experiment had the following parameters: our client tried to maintain 7000 simultaneous outbound connections and at most 8000 simultaneous connections; it waited for 48 hours before recording 1002 blocks and 100 100 transactions. In both cases, all inventories of the same type were recorded from the beginning of the recording phase to one hour after the last object was recorded. During the initial waiting phase, it used three threads to establish connections; one of them was shut down during the recording phase. In total, the experiment lasted for approximately 9 days: two for the initial waiting phase and seven for the recording phase. From our measurements, we draw results regarding the network population, block and transaction propagation. All data processing and plotting was done using Python 3.5.1, Numpy 1.11.0, Scipy 0.17.1, Scikit-learn 0.17.1, and Matplotlib 1.5.1 [WCV11; JOP+01; PVGM+11; Hun07].

A first result is the comparison of the size of the network as seen respectively by Parasolier and Bitnodes [BN]. Parasolier's dataset is the number of established Bitcoin connections at the time each block was logged; that of Bitnodes is the reported number of online routers for each available time stamp in the minimal window encompassing Parasolier's dataset. To simplify visual comparisons, we generate a third dataset by shifting Parasolier's dataset to Bitnodes' mean. The results are reported in Figure 3.1. Correlation was not quantified.

The rates of positive and negative churn (respectively join and leave operations) are compared as well. Parasolier's dataset corresponds to the cardinal of the set difference between two consecutive blocks divided by the difference of their time stamps (in seconds). Since Bitnodes already reports a positive and negative churn for each data point, these values were also divided by the difference between their assigned time stamp and the previous one. The results are reported in Figure 3.2. For each dataset, closeness of the positive and negative churn rates is tested using Student's t-test to compare the means. The Benjamini-Hochberg correction [BH95] is applied to the p-values to account for the two tests performed.

For blocks, Figure 3 from Decker *et al.*'s 2013 measurement [DW13] is reproduced

using our dataset. It empirically estimates the PDF of the reception time of a block after its first observation. In order to do so, for each block, the list of time stamps of `inv` messages announcing it was collected; the lowest value, approximating the time when the block was introduced in the network, was subtracted from each time stamp. `Inv` messages were grouped in two categories for each block: *expected* and *other*. The former corresponds to the `inv` announcing the block sent by a neighbour with which a connection had already been established when the block started propagating and maintained until the `inv` was received; the latter corresponds to every other case, *i.e.* repeated announcements from a single neighbour and messages from neighbours with which a connection was not constantly maintained over the period ranging from the time the block appeared in the network to their announcing it. All pairs of blocks at the same height in the blockchain (that is, conflictual blocks) were excluded from the data set because of their limited initial propagation. The histogram, using time steps of 0.1 s is then normalized.

Decker *et al.*'s fitted curve [DW13] (exponential distribution with parameter 0.107) is plotted for visual comparison. Additionally, we fitted two curves on the plotted portion of the histogram using the Levenberg-Marquardt non-linear least-squares [Mor78; JOP+01]: that of an exponential distribution and that of a biexponential distribution whose PDF is f_{x_0,a_1,a_2}, defined as follows:

$$f_{x_0,a_1,a_2}(x) = \begin{cases} k_1(1-e^{-a_1 x}) & \text{if } x \leqslant x_0 \\ k_2 e^{-a_2(x-x_0)} & \text{if } x \geqslant x_0 \end{cases} \text{ where } \begin{cases} k_1 = \left(x_0 + (1-e^{-a_1 x_0})(\frac{1}{a_2} - \frac{1}{a_1})\right)^{-1} \\ k_2 = k_1(1-e^{-a_1 x_0}). \end{cases}$$

This function is indeed a well-defined PDF over $[0,\infty)$ for $x_0 \geqslant 0, a_1 > 0, a_2 > 0$ such that $x_0 + (1-e^{-a_1 x_0})(1/a_2 - 1/a_1) > 0$: its integral is equal to 1 and its image is in $\mathbb{R}_+$. It is, additionally, continuous over $[0,\infty)$. We used the coefficient of determination R^2 to compare the fitness of the three curves. We report the results in Figure 3.3 where we only plot and fit curves on the restriction of the histogram to the range 0 s to 40 s.

We evaluated three aggregation statistics: the mean, the median, and the 95$^{\text{th}}$ percentile of reception times. This is done separately for the two categories previously defined and for their union. The estimation is performed for different thresholds; in each case, all reception times greater than the given threshold are discarded. The threshold set, in seconds, is $\{i \times 10^j | i \in [[1,4]], j \in [[0,5]]\} \cup \{10^6\}$. We discarded conflictual blocks as well and report the results in Figure 3.4.

Finally, the set of recorded transactions is scrutinized for double-spending attempts.

3.2.2 Results

Parasolier started running the modified Bitcoin client at 13:04:30 GMT on Monday, August 8$^{\text{th}}$, 2016. Recording started 48 hours later. 100 100 transactions had been recorded at 13:37:41 GMT on Wednesday, August 10$^{\text{th}}$, 2016 and 1002 blocks had been recorded at 06:07:08 GMT on Wednesday, August 17$^{\text{th}}$, 2016. The first block

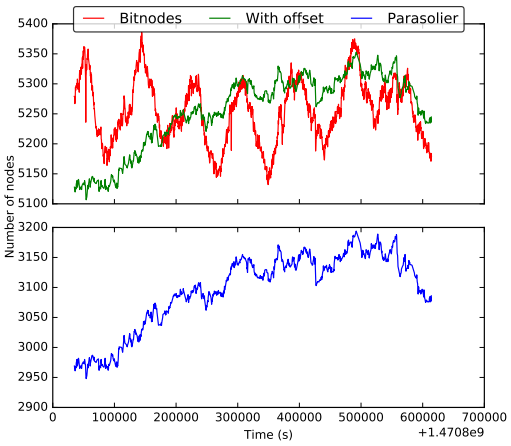

Figure 3.1: Total size of the Bitcoin network as measured by Bitnodes and Parasolier.

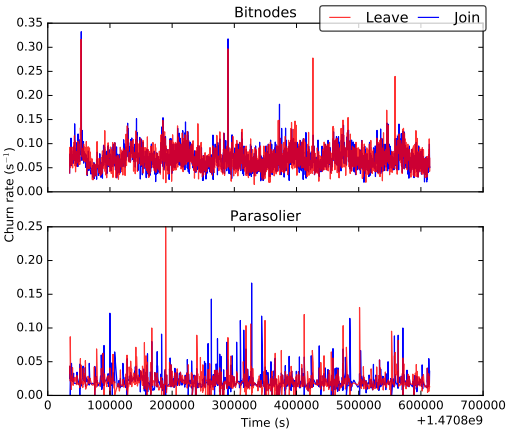

Figure 3.2: Churn rate in the Bitcoin network as measured by Bitnodes and Parasolier.

recorded was `0000000000000000040241363b56921253509e73a6d97922dced623df85e32ff`, at height 424 562 on the main chain, and the last one `00000000000000002cdf87608fe2536415c5da70db253a5aadaf72cac9ecd1f`, at height 425 560. A 1003$^{\text{rd}}$ block (`000000000000000055a3d177683aa025fb763be716ce4e812fa521406553996`, at height 423 508 and having lost a fork) was received on Monday, August 15$^{\text{th}}$, 2016 at 20:28:59 GMT but rejected as too old: its time stamp corresponds to Wednesday, August 3$^{\text{rd}}$, 2016 at 17:06:55 GMT.

Figure 3.1 represents the size of the network as a function of time. For Bitnodes, the network comprises 5255.73 routers on average, while Parasolier only maintained 3097.00 simultaneous connections on average. The difference, of 2158.72[4], represents 41.0 % of the figure advertised by Bitnodes. The peak-to-peak ranges are respectively 254 (4.7 % of the maximum value) and 246 (7.7 %).

Figure 3.2 represents the churn rates of the network as a function of time. For Bitnodes, the positive (resp. negative) churn rates has a mean of $6.94 \times 10^{-2}\,\mathrm{s}^{-1}$ (resp. $6.96 \times 10^{-2}\,\mathrm{s}^{-1}$) and a standard deviation of $2.2 \times 10^{-2}\,\mathrm{s}^{-1}$ ($2.3 \times 10^{-2}\,\mathrm{s}^{-1}$). The corrected p-value is approximately 0.834. For Parasolier, these values are respectively equal to $2.28 \times 10^{-2}\,\mathrm{s}^{-1}$ ($2.24 \times 10^{-2}\,\mathrm{s}^{-1}$), and $1.6 \times 10^{-2}\,\mathrm{s}^{-1}$ ($1.6 \times 10^{-2}\,\mathrm{s}^{-1}$), with a corrected p-value also approximately 0.834. The maximum and minimum values for positive and negative churn rates as seen by Bitnodes and Parasolier are grouped in Table 3.2.

Figure 3.3 represents the empirically estimated PDF of the propagation time of a block to a node. Over the measuring period, three forks occurred and were all resolved after only one block; thus, a total of 6 conflictual blocks were excluded from the results. This yields 2 045 716 *expected* announcements for a total of 2 996 363 announcements kept: the expected ones represent 68.3 % of them. The R^2 scores

[4] All values have been rounded to the selected decimal precision, hence the mismatch.

	Positive		Negative	
(s^{-1})	Min	Max	Min	Max
Bitnodes	0.020	0.332	0.015	0.316
Parasolier	0	0.167	0	0.250

Table 3.2: Minimum and maximum values of the churn rate in the Bitcoin network as measured by Bitnodes and Parasolier.

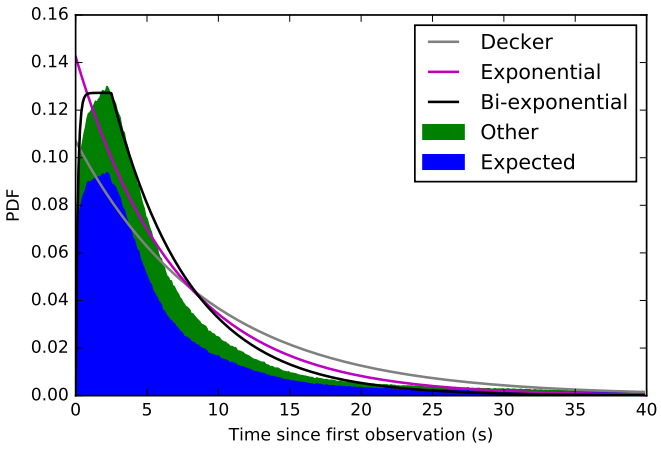

Figure 3.3: Normalised histogram of times since the first announcement of a block and fitted curves.

for the curve fitted by Decker *et al.* [DW13], our exponential curve fitted with parameter $0.142\,s^{-1}$, and our bi-exponential curve with parameters $a_1 = 6.880\,s^{-1}$, $a_2 = 0.182\,s^{-1}$, and $x_0 = 2.494\,s$ are respectively 0.857, 0.909, and 0.980. The maximum measured values for expected announcements and the whole set are respectively 139 936 s (1 d, 14 h, 52 min and 16 s) and 530 440 s (6 d, 3 h, 20 min and 40 s).

Figure 3.4 shows aggregation statistics for subsets of the recorded blocks announcements. The top figure depicts the fraction of the total set represented as a function of the chosen threshold; for the expected set, the subset contains the entire set for thresholds above 2×10^5 s; in the other two cases, the subset only equals the entire set for the last threshold, 10^6 s. However, for each set, more than 95 % of the set is included for all thresholds above 100 s. We use these values as remarkable thresholds and give numerical values for the fraction of announcements received in less time than the threshold and the mean, median, and 95^{th} percentile of the reception times of these subsets of announcements in Table 3.3.

Of the first 100 100 transactions received during the recording phase, constituting the dataset, there were only 14 446 different ones. The maximum number of times a transaction has been received is 759, its mean and standard deviation are respectively

	Expected		Other			All		
Threshold (s)	100	2×10^5	100	2×10^5	10^6	100	2×10^5	10^6
Fraction	0.98	1	0.961	0.998	1	0.968	0.999	1
Mean (s)	7.8	51.5	8.4	631.7	1379.3	8.2	396.0	840.6
Median (s)	4.0	4.1	4.3	4.5	4.5	4.2	4.3	4.3
95^{th} percentile (s)	30.0	41.7	32.5	62.5	66.8	31.5	52.3	54.1

Table 3.3: Subset of values from Figure 3.4.

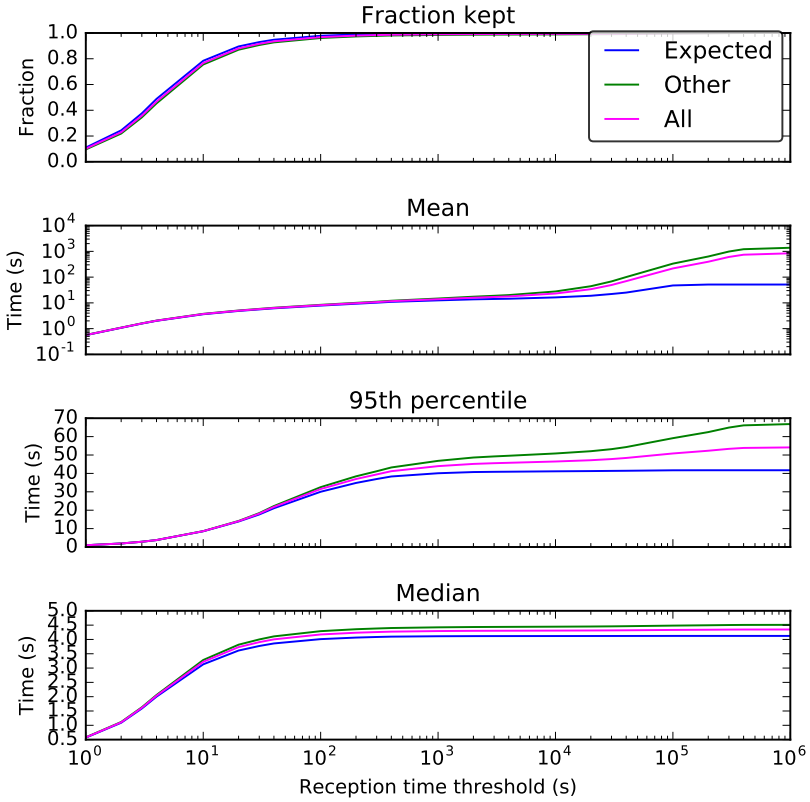

Figure 3.4: Fraction of the dataset received, mean, median and 95^{th} percentile of reception times of block announcements for various wait thresholds.

equal to 6.93 and 17.33. The set contains two pairs of double-spending attempts. In each case, the two conflicting transactions are actually the same one, signed with different ECDSA nonces: the only difference between the two transactions is the signature (but both are valid), which is enough to alter the transactions's hash and make it appear as a double-spending attempt.

3.2.3 Discussion

Let us make a number of general remarks about the overall design of the experiment before going further into the discussion of all the reported results. First, Decker*et al.*'s 2012 experiment was ten times as long as ours, which makes their results more resilient against periodic effects: the activity in the network may depend on the seasons, *e.g.* if people shut down their nodes while on holidays. We do not capture such phenomena.

Then, the accuracy of the sampling of the network could be greatly improved. Indeed, we only estimate the time stamps of the snapshots we took of Parasolier's neighbours as the time at which logging the list was actually performed, but the impact of this drift is below the order of seconds. However, more importantly, our sampling is coarse: the longest time between two snapshots in our experiment is approximately 108 minutes, the time it took the network to find block 425 380. Given that we miss all events where a neighbour disconnects and reconnects between two consecutive snapshots, our estimate of the churn is probably a lower bound. Modifying even more Core's code to actively log all connections and disconnections[5] would provide much better estimates of the churn rates. Finally, tracking neighbours over successive connections is impossible for Bitcoin: a router with the same network address as a previous neighbour may actually be a completely different device, and there are several scenarios in which two routers with the same IP address but different port numbers may or may not correspond to the same machine.

Additionally, Parasolier was using a large part of its CPU, and RAM and swap memories to maintain this many parallel Bitcoin connections and perform Core's regular operations as well as logging the experimental data. We could push the experiment further and over a longer period of time with a more complex architecture supported by more hardware: given that Parasolier recorded more than 39 GiB of data over the experiment, running again a similar experiment with a back-end database processing the data on the fly to manage the experiment would be necessary to prevent shortcomings such as our recording only a seventh of our expected number of different transactions.

Figure 3.1 roughly shows that Bitcoin's churn follows a cycle whose period is roughly equal to a day. The maximum value of each peak corresponds respectively to approximately 17:34, 19:35, 21:10, 17:14, 14:58, 19:28, and 19:25 (all hours given in GMT) on respectively Wednesday, August 10$^{\text{th}}$, 2016 to Tuesday, August 16$^{\text{th}}$. Western Europe used GMT+2 and most of America GMT-4 to GMT-7 during that

[5]Core natively logs all connection establishments and tear down but identify neighbours by a counter.

time: these peaks correspond to evenings in Europe and daytime in America. Thus, a possible explanation of this churn is that many Bitcoin routers run on computers that are online during the day in America and shut down at night. However, it still only represents a small fraction of the network as seen by Bitnodes, as the peak-to-peak range is less than 5 % of the maximum value.

A large part of the churn may also be completely invisible to Bitnodes, which can only connect to *reachable* routers: most typical home computers are located behind a NAT-box, which requires some extra configuration to enable port-forwarding and let other routers initiate connections to them. It furthermore seems likely that a fraction of those are only up when their owners are home and awake. However, determining the fraction of the network this represents, or its actual part in the churn reported by Bitnodes and Parasolier, is near impossible.

A last remark to be made about Figure 3.1 is that our 48 h initial waiting period was apparently not long enough: it took Parasolier two to three more days to enter a "permanent" regime, where its number of connections follows similar trends as that of Bitnodes. How Bitnodes manages to maintain 2000 more connections than Parasolier is unclear to us, but possible explanations are that it may run even more connection threads than we do, that it may use a different neighbour discovery protocol and that it may use several servers to increase its chances of having other routers trying to reach it, letting a centralised server aggregate the different lists of neighbours.

Figure 3.2 shows that Bitnodes' churn rate undergoes fast variations but remains contained in a window ranging from $0.05\,\mathrm{s}^{-1}$ to $0.10\,\mathrm{s}^{-1}$ with very few exceptions, while that of Parasolier has smaller variations (from $0.01\,\mathrm{s}^{-1}$ to $0.04\,\mathrm{s}^{-1}$) but many more unusual peaks. This may, however, be explained by the different sampling rate. Another surprise is that the peaks do not seem to match: the highest peak reported by Bitnodes, shortly after the beginning of the recording, corresponds to a sudden drop in the number of neighbours in Figure 3.1 but to no unusual value in Parasolier's measurements. Finally, the assumption of independence used to compute the p-values may be questioned: issues located between the recording machine and the Internet (*e.g.* in the ISP's network) will be felt by both the positive and negative churns.

An intermediate conclusion on the churn in the Bitcoin network is that the apparent size of the network was, all in all, relatively stable throughout our experiment. However, this happens because the positive and negative churn compensate each other rather than because they are non-existent. Thus, improvement proposals should take it into account.

Figure 3.3 reproduces Decker *et al.*'s curve [DW13] but shows an improvement in the propagation delays for blocks. Indeed, the exponential curve that fits best our data has a higher parameter, which indicates that the PDF is more concentrated around small reception times. We get an even better fit by using a bi-exponential curve to account for the shape of the distribution. This improvement may come from the modifications the protocol has undergone since the first study (*e.g.* transmission and validation of headers first), but also from improvements of the global Internet

infrastructure and speed, with an ever growing share of the network built on optic fibres.

Dropping repeated announcements rather than marking them as unexpected does not significantly modify the results: only 3309 announcements are in this case, which represents a mere 0.1 % of the total number of announcements used above, and the outliers are not part of them.

Figure 3.4 shows data not reported by Decker *et al.* [DW13]: the variation of aggregation statistics depending on the amount of time after which recording is stopped for a specific block. The latest block advertisement was recorded more than 6 days and 3 hours after the first corresponding announcement was received: given that the recording phase lasted slightly more than 6 days and 18 hours, it does not seem unlikely that increasing the length of the experiment would still increase the mean reception time significantly. Thus, Decker *et al.'s* assertion that the mean propagation time is 12.6 s [DW13] is not satisfying without an explicit cut-off value. The 95^{th} percentile only increases by a factor slightly below 2 over the last 3.2 % of the recorded data, and the median is relatively stable (increased by only 5 %), but the mean, more sensitive to outliers, is multiplied by 105 over the last 3.2 % of the recorded data.

Though computing the median reception time on the lowest 95 % recorded values is equal to computing the 47.5^{th} percentile on the whole dataset, what we do is conceptually different: we compute it for a fixed threshold. Indeed, increasing the duration of the experiment to receive more very late advertisements would not modify the values we report for all thresholds below 3600 seconds, the minimum time each advertisement was allocated to reach Parasolier. Instead, it would affect the fraction of the dataset represented by those values. At this point, we need to point out that our experiment was already somewhat unfair, as each block was allocated less time than the previous one to propagate in the network before the end of the experiment. A fairer experiment design would take that into account; we did not actually expect to receive advertisements for blocks that had started to propagate more than one hour ago, since connection establishment is normally used to determine which blocks to announce to the new neighbour. This could actually be used to improve further the block propagation detection mechanism that we have implemented based solely on advertisements using `inv` messages (see Appendix C.3), which prove somewhat unreliable, possibly in part because of upload limits respected by some routers: they may wait a long time before sending announcements once their daily or weekly upload quota is reached. However, given that even expected announcements may arrive very late, it would potentially not completely fix the situation.

Chapter 4

Improving Bitcoin

The last part of this work focuses on improving Bitcoin. It does so in two ways: first, Section 4.1 describes a network simulator that we implemented in order to test efficiently possible solutions to some of Bitcoin's problems. Then, Section 4.2 presents an approach based on distributed hash tables (DHTs) to enhance Bitcoin's safety property.

4.1 Network simulator

Bitcoin implements several alternate blockchains and networks on which experiments can be conducted either in open or closed environments. However, evaluating the effect of some parameters on the overall system can be quite challenging in those environments for various reasons, including concurrent experiments and size differences. Thus, we chose to implement a simulator that replicates the parts of Bitcoin that are of interest to us, in order to be able to analyse the effect of protocol changes on the network. Section 4.1.1 describes our implementation and the protocol we followed to validate it. Then, Section 4.1.2 presents the results of the validation experiment. Finally, Section 4.1.3 highlights possible use cases.

4.1.1 Method

Our simulation focuses on blockchain replication over the network: the main part of Bitcoin that is implemented is the block propagation mechanism. The blockchains branches present in the network are tracked, along with their acceptance by nodes. The implementation is in Java 1.7.0_101. All parameters described as *simulation parameter* in the following are to be fine-tuned through an experiment made to validate our model simply called the validated experiment. On the other hand, so-called arbitrary parameters are assigned a value or a range thereof based on the literature and were not evaluated individually. A single (pseudo-)random number generator was used for the experiment. It was seeded with the arbitrarily chosen value of `123584352` to yield reproducible results.

We define three categories of nodes: regular, jumbo and NATed. The first one correspond to what constitutes most of Bitcoin's network according to the authors of [MLPG+15]: nodes following the usual connection protocol, they establish 8 outbound connections and accept up to 117 inbound ones. The second one refers to very well-connected nodes, such as those usually maintained by Bitcoin trackers, accepting an infinite amount of inbound connections and establishing an arbitrary number of outbound ones. Finally, the third category regroups all the nodes hidden behind firewalls, establishing 8 outbound connections and refusing all inbound ones. Nodes can receive blocks from their neighbours, validate them and finally broadcast them. They can also find blocks, with a probability equal to their share of the total hashing power of the network.

The simulator uses discrete time steps of arbitrary length. At each step, all nodes validate their buffered, newly-received blocks and broadcast the recently validated ones. Nodes can also leave an join the network, and reestablish outbound connections when they have less than their target number.

The number of steps required to validate a block is fixed for each node, drawn from a Gaussian distribution whose standard deviation is 1 and whose mean is a parameter of the simulation. To that is added the time needed to receive the block. Only one block can be received at the same time, but a block can be received while another one is being validated.

The broadcast model is as close to that of Bitcoin as possible: each node maintains a send buffer for each of its neighbours and iterates over them to find non-empty ones. When it finds one, it flushes it and sends all blocks the associated neighbour is interested in. The time it takes to send a single block and the maximum number of non-empty buffers a node iterates over in a time step are simulation parameters. The sending process is batched: nodes wait to have sent all the blocks they have started sending during the same time step before resuming the looped iteration over the buffers to send a new batch.

Finally, join and leave operations use the probabilities derived from Bitnodes; several events of each kind can happen during the same time step. Nodes establish as many connections as needed to fill up their targets in a single time step; leaves are performed before joins, themselves performed before the nodes already in the system compensate for their lost connections. When a new node join, it catches up a number of blocks chosen uniformly at random between 1 and an arbitrary value.

To validate the simulator, we tuned it to replicate the results from the measure of block propagation times described in Section 3.2. We considered a network with 5000 regular nodes, 50 jumbo ones and 1000 NATed ones, as an approximation guided by Bitnodes [BN] and the results from [MLPG+15]. Jumbo nodes each established a number of outbound connections chosen uniformly at random between 90 and 700. Another node, called the *measuring* one, replicated what Parasolier did during our experiment: connect to as many nodes as it could and log the arrival time of block announcements. Time steps each lasted 0.1 s, and a block was found by a randomly selected node (apart from the measuring one) every 1000 time steps. New nodes had to catch up with at most 5 blocks. A grid search over mean validation and transmis-

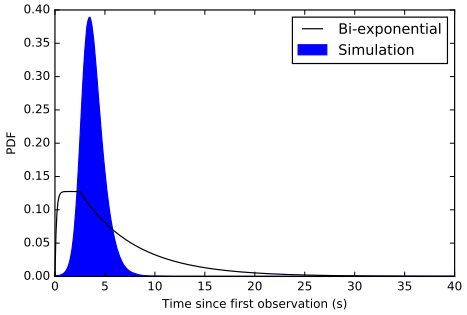

Figure 4.1: Normalised histogram of times since the first announcement of a block in the simulation with mean validation time of 0.1 s, transmission time of 0.7 s, and at most 1 simultaneous block transmission.

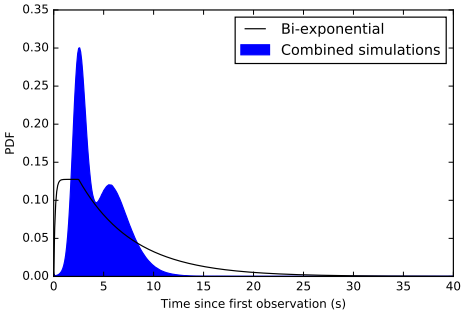

Figure 4.2: Normalised histogram of times since the first announcement of a block in the combined simulations with mean validation time of 0.1 s and 0.6 s, transmission time of 0.2 s and 1 s, and at most 1 simultaneous block transmission.

sion times ranging from 1 to 10 time steps (both included) and maximum number of simultaneous broadcasts ranging from 1 to 4 (both included) was performed to determine their best values. An histogram similar to that of Figure 3.3 was plotted (without distinguishing expected announcements from the rest); the evaluation was performed by computing the R^2 score of the bi-exponential curve with the parameters from that same figure. A total of 5000 blocks were released. Every 50 blocks, the graph was fully regenerated to average out strange configurations.

To simulate some of the heterogeneity of the graph (different block sizes and nodes,...), we also perform the same evaluation over pairs of histogram: for each such pair, we average the two estimated PDFs to obtain a third one that we compare to the fitted bi-exponential curve.

4.1.2 Results

Figure 4.1 represents the simulated PDF of the propagation time of a block to a node best fitted by our bi-exponential curve with the parameters from Section 3.2.2. The parameters corresponding to that simulation are a mean validation time of 0.1 s, a transmission time of 0.7 s and each node could send at most 1 block at a time. The R^2 score for our bi-exponential curve is 0.4025. The latest announcement was received after a simulated time of 75.8 s.

Figure 4.2 represents the same graph where instead of selecting the best experiment, we select the best average of two experiments. The parameters corresponding to those simulations are a mean validation time of 0.1 s and 0.6 s respectively, a transmission time of 0.3 s and 1 s respectively and both only admit 1 simultaneous block transmission per node. The R^2 score for our bi-exponential curve is 0.6122.

4.1.3 Discussion

A number of results are somewhat unsurprising in this simulation. Indeed, the time between the first and last announcements of a given block is much shorter than in the Bitcoin network, but what is really surprising in that comparison is rather the enormous delay in the Bitcoin network: all of our simulated nodes behave as expected and the propagation of any message in the network terminates somewhat quickly.

Similarly, the shape of the simulated PDF corresponds to the usual three-phased propagation process, with a very slow start, an exponential progress once enough nodes have started propagating and finally a slow termination. Given our deterministic propagation mechanism (except for the churn in the network), the final phase is rather efficient.

Here again, what is really surprising is that this simulates extremely poorly the measured behaviour of the Bitcoin network: its "slow-start" phase is seemingly non-existent and it reaches almost immediately a quite fast expansion phase that also quickly gives turn to a very slow final phase, even though the nodes are supposed to simply iterate in a completely deterministic way over their neighbours to broadcast data. There are many apparent explanations for this difference in behaviour. First, in the simulation, nodes detect instantly when one of their neighbours disconnects, and re-establish instantly a connection; Bitcoin is said to be much less efficient in that regard but the detection of errors thrown by the TCP sockets used to handle the connections may actually suffice to void this argument. Then, only block data is simulated: the Bitcoin network is used to transmit many more messages such as transactions and addresses. Added to the random delays which can arise over the Internet, this may partially explain Bitcoin's longer tail. Bitcoin probably also has a much larger variance because of the variety of nodes in the network, that our model with only 3 types does not fully capture.

The variable size of blocks was approximated in Figure 4.2: it behaves as if there were two types of blocks, small and large ones. It greatly increases the fitness of the bi-exponential curve (by 54 %). A better fit could probably be obtained by combining even more simulations and possibly computing a weighted average of the PDFs to account for the unequal distribution of blocks.

Finally, given how much more efficient the simulated network quickly becomes, the inability of our model to capture this very quick initial phase really stands out. Our main conjecture to explain this failure is the unforeseen importance of parallel, more efficient communication means, such as the Fast Relay Network or its successor [Cor16]. Through these specialised network, blocks seem to be almost instantly transmitted to sufficiently many nodes to skip the slow start phase of the usual gossip mechanism in random graphs.

4.2 Reinforcing Bitcoin's safety

In Section 2.2, we have presented Bitcoin's liveness, safety and validity properties. However, we consider the safety not to be strong enough for a financial system. This section presents and extends our work from [LAL], in which we have showed how to enhance Bitcoin's safety to ensure the following:

Property 5 (Bitcoin's strong safety)
A transaction confirmed by some rational node will eventually be deeply confirmed by all rational nodes at the same height in the blockchain.

In order to achieve that result, we need to add a few more assumptions to our model. First, the maximum number of Byzantine nodes in Π at any time is set to $f = \lfloor(|\Pi| - 1)/(3 + \epsilon)\rfloor$, for some $\epsilon > 0$; this bound derives from the underlying partially synchronous network [DLS88].

We insist on the assumed absence of hash collisions: transaction, block and outputs are uniquely defined by their 256-hash; we denote by $h(\cdot)$ the function yielding the 256-hash of transactions, blocks, outputs and extend its definition so that for an input i consuming output o_i, $h(i) = h(o_i)$. We further assume that these hashes are uniformly distributed in $\{0,1\}^{256}$, as could be expected from a standardized hash function. We call $h(\theta)$ the *ID* of object θ

The uniform distribution seems to hold in practice, as shown in Figure 4.3. It depicts the frequency at which each hexadecimal character appears as the first of $h(x)$, for x iterating over the set of transactions contained in 100 consecutive blocks starting at height 420 000 and their inputs. 19 transactions were excluded because they were too big to be decoded by Core's RPC API; thus, this study covers 368 327 hash results over 102 283 transactions. The dashed line represents the mean, equal to 0.0625 as expected from the uniform distribution. The low standard deviation of $4.78e^{-4}$ (with Bessel's correction) confirms the good performances of the hash function as regards the pseudo-randomness of the output.

4.2.1 Conflict Detection Services

Figure 4.4 depicts the path of transactions from the users to the blockchain: currently, users submit transactions to a transaction conflict detection service (TCDS) made of the routers, stores and wallets; in case of conflict, each node p decides locally which of the transaction should be accepted, based mostly on which one provides the highest expected profit for p. Node p then mined to confirm the transaction and sends it to other routers, which again decide individually whether to accept it. The process is then similar for blocks from the miners to the blockchain through the block conflict detection service (BCDS).

The above mechanism, chosen for performance reasons, yields an inconsistent validation of conflicting transactions and blocks. This sections describes our synchronization mechanism forcing these two conflict detection services (CDSs) to provide the same answer to each node.

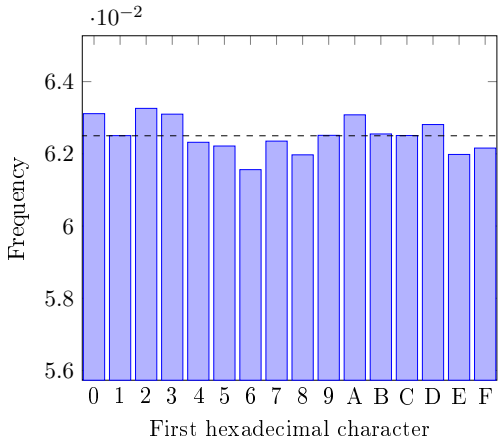

Figure 4.3: Distribution of the first character of the hashes of transactions and inputs over $\{0,1\}^4$.

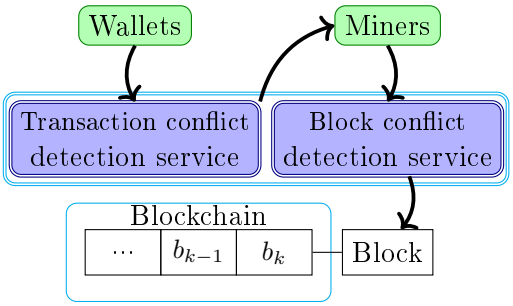

Figure 4.4: Orchestration of Bitcoin: wallets submit transactions to the network. Once validated, the miners include them in the blocks they build which, once validated, are accepted in the blockchains.

Since the blockchains is, at its core, a simple distributed database, an analogy can easily be derived between emitting transactions and writing in the database. Thus, what we need is a process to grant exclusive access to inputs to the transactions that use them in order to prevent double-spending attempts: transactions need to explicitly lock their inputs. Yet, unless care is taken, locking objects one by one may cause deadlocks. As the application we consider involves different entities spread over a large area, it is not advisable to rely on having all of them conform to the same locking strategies. Moreover, from a performance viewpoint, it may be impossible to run deadlock detection and prevention protocols assuming independent object locking.

The three works studied in [ALLS16] failed to improve Bitcoin's overall security because they all introduce single points of failure in the form of $\mathcal{E}_\ell$, tasked with the management of all locks for the entire system. We aim at avoiding this pitfall by introducing the least amount of synchronization required to guarantee consistent conflict resolutions for both transactions and blocks.

4.2.1.1 Specification of the CDSs

Transaction Conflict Detection Service The purpose of the TCDS is to ensure that concurrent transactions do not try to use common inputs. We propose a TCDS that provides the equivalent of an atomic locking mechanism for all of the inputs of each transaction. Formally, the TCDS offers a single method called `grantInputs`. It accepts a transaction T as parameter and returns with GRANTED or DENIED. When an invocation returns with GRANTED, we say that *the method exclusively grants the inputs in I_T to T* or, in short, that *T is granted*. Conversely, T is *denied* when

`grantInputs`(T) returns DENIED.

Based on this definition, we require the TCDS to provide the following properties:

Safety: If a transaction T is granted then no other transaction T' such that $I_T \cap I_{T'} \neq \emptyset$ is granted.

Liveness: Each invocation of `grantInputs` eventually returns.

Non triviality: If there exists an invocation of `grantInputs` on $T \in \mathcal{T}$, and no other invocation of the `grantInputs` on $T' \in \mathcal{T}$ such that $I_T \cap I_{T'} \neq \emptyset$, then T is granted.

Block Conflict Detection Service The BCDS aims at ensuring that any validated block has at most one valid block as its immediate successor. It offers a single method, `grantBlock`, that accepts a block b as parameter. This method returns with GRANTED or DENIED. When an invocation returns with GRANTED, we say that *the method validates* b as the unique successor of $p(b)$ or, in short, that b *is granted*. Conversely, b is *denied* when `grantBlock`(b) returns DENIED. Based on this definition, we require the BCDS to provide the following properties:

Safety: If a block b is granted then no other block b' such that $p(b) = p(b')$ is granted.

Liveness: Each invocation of `grantBlock` eventually returns.

Non triviality: If there exists an invocation of `grantBlock` on $b \in \mathcal{B}$ and no other invocation of `grantBlock` on $b' \in \mathcal{B}$ such that $p(b) = p(b')$ has ever been granted, then b is granted.

With such a system, forks are prevented and transactions may be considered deeply confirmed as soon as they are included in a granted block.

4.2.1.2 Implementation of the CDSs

We propose to distribute the implementation of the CDSs over specific sets of nodes randomly chosen in the system. This section supposes that each node has a (not necessarily unique) identity. Each object θ (*i.e.*, input, transaction or block) is assigned a *referee* π_θ, the node whose identity is the closest to $h(\theta)$.

When a wallet creates a transaction T, it submits it to its referee π_T, which is in charge of invoking the TCDS for T. This invocation consists for π_T in asking a lock to the referee of each input in I_T, in the lexicographical order of the input IDs. If any such lock is denied, π_T might try up to some threshold of times; if it fails to obtain the lock afterwards, it releases all previously obtained locks and returns DENIED. Otherwise, after obtaining all the locks, π_T returns GRANTED. The `Release` method consists in proving to the referee of each locked input that transaction T will be denied by exhibiting the conflicting transaction T' that was

GRANTED. The correctness and, in particular, the lack of deadlocks, result from the fact that locks are always obtained in lexicographical order. A lock can be implemented using a combination of Test-and-Set and Reset primitives. The referee π_i that wishes to lock input i first tests the value of a binary register. When this value is 0, it modifies the register to 1 and uses the lock. Releasing a lock is done by resetting to 0 the register value. The fact that T has been granted the lock on i is proven by π_i's signing T; the fact that T has been granted is proven by π_T's signature of T. Each signature is bundled with the identity of the signer so that any node can verify both that the signature is correct and that the signer was the appropriate referee.

Bitcoin can easily be extended to accommodate this process: each transaction T must include a special *validation* output o_{val}; π_T can then compute a group signature (*e.g.* [Bol03]) using those of each input referee and its own and append it, along with everything needed to verify it, to the challenge $\chi_{o_{\text{val}}}$. The value $v(o_{\text{val}})$, called the *validation fee*, provides an incentive for referees. A fair and easy way to share the output is to randomly pick one of the referees and give it the entire reward. This requires seeding a random number generator in a publicly verifiable way, and referees should not be able to manipulate the draw; using some information that can only be published after the TCDS has returned, *e.g.* the block in which the transaction is included, can achieve this. Thus, giving transaction T's validation fee to its k-th referee, where $k = h(h(T)||h(b)) \bmod s$ with $b \in \mathcal{B}$ such that $T \in c(b)$ and s the total number of referees is a possible solution. Finally, any node can verify that a transaction T was granted by checking that $\chi_{o_{\text{val}}}$ contains the signatures added by the referee π_T and that they are correct. This process leads to the fact that transactions now each have two IDs: one used for the TCDS operations (defining the referee and verifying its signature), and one used to refer to the transaction once it has been granted.

The process is simpler for blocks, since each one only has one parent to lock. When a miner generates a block b, it can submit it to $\pi_{p(b)}$, in charge of invoking the BCDS on b. To simplify the implementation, $\pi_{p(b)}$ can mark that the BCDS returns GRANTED by applying the mechanism used by the TCDS on the coinbase transaction of b; since coinbase transactions have no inputs and can only be propagated as part of a block, they do not need to be granted anyway. The remark about the two different IDs holds as well, with the added remark that the hash used in the PoW does not cover the referee's signature.

4.2.2 Leveraging DHTs to Implement the CDSs

The fundamental principle of our two CDSs is the link between each Bitcoin object (*i.e.* transaction, input, and block) and its referee. Thus, each transaction is granted an exclusive access on each of its inputs and each block has at any time at most one successor. Our solution to implement such a link simply consists in bringing some structure to the underlying unstructured peer-to-peer overlay of Bitcoin. The topology of unstructured overlays conforms random graphs, *i.e.* connections between

nodes are mostly established according to a random process and routing is not constrained. Object placement enjoy the same absence of constraints: Bitcoin uses flooding techniques to let each node retrieve objects. On the other hand, structured overlays, also called *DHTs*, build their topology according to structured graphs. For most of them, the identifier space is partitioned among all the nodes of the overlay. Nodes self-organize within the graph according to a distance function based on identities (*e.g.* two nodes are neighbours if their identities share some common prefix), and possibly other criteria such as geographical distance. Each application-specific object is assigned a unique ID selected from the same identifier space. Each node owns a fraction of all the object of the system. The mapping derives from the distance function.

Any DHT could be a valuable candidate to organize nodes and objects in Bitcoin, as long as the chosen DHT is capable of handling churn (see *e.g.* [HKZG15] or Section 3.2) and the presence of colluding Byzantine nodes. S-Chord [FSY05] and PeerCube [ALRB08] are two such DHTs. Briefly, both DHTs gather nodes into clusters, each constituted a vertex of the graph. All the routing and storage operations classically devoted to each node in a non-clustered DHT are jointly handled by all the nodes in a cluster, through Byzantine-tolerant consensus protocols. This makes such DHTs highly resilient. In addition, the impact of churn is mainly handled at cluster level, which minimizes the impact on the graph structure of the DHT. Finally, both DHTs limit the sojourn time of nodes at the same position of the overlay (through induced churn [AS04]) to prevent the adversary from choosing its own positions and eclipsing correct nodes from a given region of the overlay. Thus, each of our referees actually corresponds to a cluster of nodes, which guarantees its safety.

Despite their qualities, both DHTs assume the presence of a trusted third-party to act as a public key infrastructure (PKI) in charge of assigning certified identities to each node. Such an assumption is unrealistic in large scale, dynamic and open systems, and thus we can only rely on nodes to create themselves their identities. There is however no guarantee that each one will create a single identity, if it is profitable to get several of them. To drastically limit the number of identities per node, we leverage the PoW mechanism: each node must solve a computationally expensive challenge to create each identity, which in expectation makes the number of identities a node can maintain proportional to its computing power. Such an approach is not new [LNBZ+15]. Hence, an identity $\mathcal{I}$ comprises a public key $\mathrm{PK}_{\mathcal{I}}$, a time stamp $t_{\mathcal{I}}$, a nonce $\nu_{\mathcal{I}}$, and the hash of the last known block $h(b_{\mathcal{I}})$ of the blockchain. The public key authenticates messages. The time stamp forces an induced churn: identities have a lifetime of Δ time units, after which they expire. Finally, the nonce is used in the PoW mechanism: $\mathcal{I}$ is only considered valid if, besides not being expired, $h(\mathrm{PK}_{\mathcal{I}}||t_{\mathcal{I}}||\nu_{\mathcal{I}}||h(b_{\mathcal{I}})) < \gamma$, where γ is a network-specified target.

Time stamps cannot be trusted: a Byzantine node could either spend months pre-computing a lot of identities all with the same time stamp to flood the system and take control over a large part of it at a predefined instant, or could simply be set in the future to extend an identity's lifetime. The latter attack is mitigated

by Bitcoin's time stamp validity check: if the time stamp is too far away in the future, nodes consider it invalid. In the former case, the attack is more complex to defeat, because one cannot know *a posteriori* that the identity was precomputed. This explains the presence of a recent piece of data shared by the network, *i.e.* the hash $h(b_{\mathcal{I}})$ of the last block present in the blockchain. In order to cope with the propagation delays and transiently different local views of the blockchain, the hash of one of the last β blocks is sufficient: a larger β gives an attacker more time to precompute identities but requires less synchronization from the network.

There is no guarantee on the actual number of identities under the control of any node from Π. This explains why we only require that $f/n \leqslant 1/(3+\epsilon)$ for some $\epsilon > 0$. A precise analysis is left for future work.

4.2.3 Discussion

We now highlight some positive and negative side effects of our proposal.

4.2.3.1 Positive Impact on Adversarial Mining

The main goal of this proposal is to prevent outputs and blocks from having more than one successor each. A positive side-effect is that it also prevents some forms of adversarial mining: SPV mining and selfish mining. Indeed, the pointer to a block which is included in the header of its tentative successors is the hash covering its referee's signature. Thus, it becomes pointless to keep newly found blocks secret. Similarly, getting a newly found block from its miners to bypass the regular flooding mechanism is not enough, as even the successful miner cannot determine the final hash of the block before it is granted by the BCDS.

4.2.3.2 Negative Impact on Nodes with Weak Computing Power

It may happen that for some reasons, nodes cannot spend momentarily the computing power to create an identity. This does not jeopardize their participation to the network, in the sense that they can continue to receive blocks and locally manage the blockchain. However, during the time they do not possess an identity they cannot participate to the CDSs, and thus cannot receive fees for that. Another issue concerns the equilibrium that need to be reached between the two resource-consuming PoWs, based on their respective expected profits, and the fact that an attacker may be able to leverage this equilibrium to gain power.

For example, transactions fees currently represent only a very small fraction of the total block reward (from block 428 939 to 428 944 included, fees represent on average 3.81 % of the value of the coinbase transaction [BC.I]). Thus, most rational nodes may end up mining blocks, leaving the identity generation process vulnerable to easy 51 %-takeovers, allowing the attacker to perform double-spending attacks and reject blocks mined by others. On the other hand, it may also encourage drop-out miners (that left because it was too difficult to be profitable) to join back the identity generation process, increasing the total rational computing power.

An alternative to using PoW is to rely on PoS schemes for blocks: it focuses all the computing power on the identity generation process and prevents attackers from taking advantage of the equilibrium. However, PoS schemes need all nodes to use the same seed for the random number generator used to elect a leader, whereas PoWs can be used offline. Thus, it would make sense to use PoWs to generate identities and PoSs for blocks, assuming the existence of a secure and usable PoS scheme, solving the issues faced *e.g.* by the proposition of Bentov *et al.* [BGM14] (analysed in Section 3.1.6.3).

4.2.3.3 Scalability

Bitcoin is already criticized for its lack of scalability: the size size and generation time of blocks are such that Bitcoin can only process around 7 transactions per second [LNBZ+15]. By adding an output to each transaction, we may worsen the situation. Indeed, identities are 73 B using a 33 B compressed ECDSA public key and a 4 B nonce, while inputs typically are around 150 B (*i.e.* 40 B for the reference to the spent output and a sequence number, 33 B for the public key, 71 B in average for the signature and a few script operators). Given that, for each input, we add the group signature of the referee cluster, it requires for each signing identity to be included as well. This may double the size of a transaction. On the other hand, since forks and adversarial mining techniques are prevented, blocks could be generated at a faster pace without jeopardizing the system security.

4.2.3.4 Relaxed TCDS

Our TCDS may enforce a safety property that is too strict for Bitcoin: there is a risk that a transaction will be granted its inputs but never be included in a block because *e.g.* its fee is considered too low by the miners. This would result in money leaks, as the unconfirmed transactions would eventually be forgotten by the network and their outputs never become spendable. To circumvent this issue, leases can be used: when a lock is granted to a given transaction, it is granted only fora given duration. If the transaction wishes to use the lock for a longer period, it must revalidate its ownership of the lock before it expires. Failure to revalidate a lock is implicitly translated into a release of that lock if another transaction is trying to obtain it.

Conclusion

In this work, we have progressed towards a better understanding of the Bitcoin system: we have defined a formal model to serve as a general framework for studies of the system. We have used this model to derive Bitcoin's fundamental properties of *liveness*, *safety* and *validity*. Then, we have describe Bitcoin's current situation through a detailed analysis of some of the most prominent academic works that have been conducted on the topic since 2008, which include measurement campaigns and analyses of the underlying blockchain protocol or the financial application built upon it, their vulnerabilities and ways to fix them. We have verified some of this results through our measurement campaign and analysed its shortcomings. Finally, we have implemented a simulator in order to test quickly and cheaply improvement proposals; our difficulties in fine-tuning it have led us to the conjecture that a significant part of Bitcoin's flooding mechanism is actually performed outside of its network. We have nonetheless described our own improvement proposal to reinforce Bitcoin's safety property and make it much more usable for fast payments; in the process, it also improves the fairness of the mining process.

However, given how complex Bitcoin's ecosystem is, there are many more paths that future work can explore. Indeed, though we have discussed the theoretical feasibility of our improvement proposal, we have not yet implemented it to verify its actual scalability. A number of open questions remain, particularly in the optimal values of security parameters to achieve the expected level of security or in the formal comparison of possible alternatives such as the proof used in blocks. Many aspects of the ecosystem, such as cryptography and privacy, have been left mostly untouched without verifying if the current solutions, such as Zerocoin [MGGR13], are satisfying as regards their goals and usability.

Bibliography

[AKRSC13] Elli Androulaki, Ghassan O. Karame, Marc Roeschlin, Tobias Scherer, and Srdjan Capkun. "Evaluating User Privacy in Bitcoin". In: *Financial Cryptography and Data Security: 17th International Conference*. Springer, Apr. 2013. DOI: 10.1007/978-3-642-39884-1_4.

[ALLS16] Emmanuelle Anceaume, Thibaut Lajoie-Mazenc, Romaric Ludinard, and Bruno Sericola. "Safety Analysis of Bitcoin Improvement Proposals". In: *15th IEEE International Symposium on Network Computing and Applications (NCA)*. Oct. 2016.

[ALRB08] Emmanuelle Anceaume, Romaric Ludinard, Aina Ravoaja, and Francisco Brasileiro. "PeerCube: A Hypercube-Based P2P Overlay Robust against Collusion and Churn". In: *2008 Second IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO)*. Oct. 2008, pp. 15–24. DOI: 10.1109/SASO.2008.44.

[Ant14] Andreas Antonopoulos. *Mastering Bitcoin*. O'Reilly Media, Dec. 2014.

[ANV13] Shaikshakeel Ahamad, Madhusoodhnan Nair, and Biju Varghese. "A Survey on Crypto Currencies". In: *Proceedings of the International Conference on Advances in Computer Science (AETACS)*. 2013.

[AS04] Baruch Awerbuch and Christian Scheideler. "Group Spreading: A Protocol for Provably Secure Distributed Name Service". In: *Automata, Languages and Programming: 31st International Colloquium (ICALP)*. Springer, July 2004. DOI: 10.1007/978-3-540-27836-8_18.

[BA99] Albert-László Barabási and Réka Albert. "Emergence of Scaling in Random Networks". In: *Science* 286.5439 (1999), pp. 509–512. DOI: 10.1126/science.286.5439.509.

[Bac97] Adam Back. *Hashcash*. http://cypherspace.org/hashcash. Accessed August 1st, 2016. May 1997.

[BAF] *Bitcoin Affiliate Network*. https://www.bitcoinaffiliatenetwork.com/. Accessed July 12th, 2016.

[BC.I] *Blockchain.info*. https://blockchain.info. Accessed July 12th, 2016.

[BDEWW13] Tobias Bamert, Christian Decker, Lennart Elsen, Roger Wattenhofer, and Samuel Welten. "Have a Snack, Pay with Bitcoins". In: *IEEE P2P 2013 Proceedings*. Sept. 2013, pp. 1–5. DOI: 10.1109/P2P.2013.6688717.

[BDOZ11] Moshe Babaioff, Shahar Dobzinski, Sigal Oren, and Aviv Zohar. "On Bitcoin and Red Balloons". In: *SIGecom Exchanges* 10.3 (Dec. 2011), pp. 5–9. ISSN: 1551-9031. DOI: 10.1145/2325702.2325704.

[BF] *Bitcoin Forum*. https://bitcointalk.org. Accessed July 20th, 2016.

[BGM14] Iddo Bentov, Ariel Gabizon, and Alex Mizrahi. "Cryptocurrencies without Proof of Work". In: *ArXiv* abs/1406.5694v8 (July 2014).

[BH95] Yoav Benjamini and Yosef Hochberg. "Controlling the False Discovery Rate: a Practical and Powerful Approach to Multiple Testing". In: *Journal of the Royal Statistical Society. Series B (Methodological)* 57.1 (1995), pp. 289–300. ISSN: 00359246. URL: http://www.jstor.org/stable/2346101.

[BIP9] Pieter Wuille, Peter Todd, Greg Maxwell, and Russel Rusty. *BIP 9: Version bits with timeout and delay*. https://github.com/bitcoin/bips/blob/master/bip-0009.mediawiki. Unpublished, accessed August 9th, 2016 as a draft. 2015-2016.

[BKP14] Alex Biryukov, Dmitry Khovratovich, and Ivan Pustogarov. "Deanonymisation of Clients in Bitcoin P2P Network". In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2014, pp. 15–29. DOI: 10.1145/2660267.2660379.

[Blo70] Burton H. Bloom. "Space/Time Trade-offs in Hash Coding with Allowable Errors". In: *Communications of the ACM* 13.7 (July 1970), pp. 422–426. DOI: 10.1145/362686.362692.

[BN] *Bitnodes*. https://bitnodes.21.co. Accessed July 12th, 2016.

[Bol03] Alexandra Boldyreva. "Threshold Signatures, Multisignatures and Blind Signatures Based on the Gap-Diffie-Hellman-Group Signature Scheme". In: *6th International Workshop on Practice and Theory in Public Key Cryptography*. Public Key Cryptography (PKC). Springer, 2003, pp. 31–46. DOI: 10.1007/3-540-36288-6_3.

[BR05] Béla Bollobás and Oliver Riordan. "Slow Emergence of the Giant Component in the Growing m-Out Graph". In: *Random Structures & Algorithms* 27.1 (2005), pp. 1–24. DOI: 10.1002/rsa.20060.

[B.SE] *Bitcoin Stack Exchange*. https://bitcoin.stackexchange.com. Accessed July 3rd, 2016.

[BT] *Blocktrail*. https://www.blocktrail.com. Accessed October 18th, 2016.

[But14] Vitalik Buterin. "A next-generation smart contract and decentralized application platform". In: *White paper* (2014).

[BW] *Bitcoin Wiki*. `https://en.bitcoin.it/wiki`. Accessed August 9th, 2016.

[BW14] Conrad Barski and Chris Wilmer. *Bitcoin for the befuddled*. No Starch Press, 2014.

[BW.MP] *Comparison of mining pools*. `https://en.bitcoin.it/wiki/Comparison_of_mining_pools`. Accessed August 30th, 2016.

[Caf16] Grace Caffyn. *Bitcoin Pizza Day: Celebrating the Pizzas Bought for 10,000 BTC*. `http://www.coindesk.com/bitcoin-pizza-day-celebrating-pizza-bought-10000-btc/`. Accessed October 17th, 2016. 2016.

[Caw14] Daniel Cawrey. *What Are Bitcoin Nodes and Why Do We Need Them*. `http://www.coindesk.com/bitcoin-nodes/need/`. Accessed October 25th, 2016. 2014.

[CDEG+16] Kyle Croman, Christian Decker, Ittay Eyal, Adem Efe Gencer, Ari Juels, Ahmed Kosba, Andrew Miller, Prateek Saxena, Elaine Shi, Emin Gün Sirer, Dawn Song, and Roger Wattenhofer. "On Scaling Decentralized Blockchains". In: *Proceedings of the 3rd Workshop on Bitcoin and Blockchain Research*. 2016.

[Cor16] Matt Corallo. *The Future of The Bitcoin Relay Network(s)*. `http://bluematt.bitcoin.ninja/2016/07/07/relay-networks/`. Accessed September 27th, 2016. July 2016.

[Core] The Bitcoin Core developpers. *Bitcoin Core*. `https://github.com/bitcoin`. Commits labelled v0.12.1 and v0.13.0rc1 used as points of reference. 2016.

[DBP96] Hans Dobbertin, Antoon Bosselaers, and Bart Preneel. "RIPEMD-160: A strengthened version of RIPEMD". In: *Fast Software Encryption: Third International Workshop*. Springer, 1996, pp. 71–82. DOI: `10.1007/3-540-60865-6_44`.

[DF14] Primavera De Filippi. "Bitcoin: A Regulatory Nightmare to a Libertarian Dream". In: *Internet Policy Review* (May 2014).

[DLS88] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. "Consensus in the Presence of Partial Synchrony". In: *Journal of the ACM* 35.2 (Apr. 1988), pp. 288–323. DOI: `10.1145/42282.42283`.

[DMS04] Roger Dingledine, Nick Mathewson, and Paul Syverson. "Tor: The Second-generation Onion Router". In: *Proceedings of the 13th Conference on USENIX Security Symposium (SSYM) - Volume 13*. USENIX Association, 2004.

[DN92] Cynthia Dwork and Moni Naor. "Pricing via Processing or Combatting Junk Mail". In: *12th Annual International Cryptology Conference*. Advances in Cryptology (CRYPTO). Springer, 1992, pp. 139–147. DOI: 10.1007/3-540-48071-4_10.

[Doc] *Bitcoin: Developer documentation*. https://dev.visucore.com/bitcoin/doxygen/. Accessed August 2nd, 2016.

[DSW16] Christian Decker, Jochen Seidel, and Roger Wattenhofer. "Bitcoin Meets Strong Consistency". In: *Proceedings of the 17th International Conference on Distributed Computing and Networking (ICDCN)*. ACM, 2016, 13:1–13:10. DOI: 10.1145/2833312.2833321.

[DW13] Christian Decker and Roger Wattenhofer. "Information Propagation in the Bitcoin Network". In: *IEEE P2P 2013 Proceedings*. Sept. 2013, pp. 1–10. DOI: 10.1109/P2P.2013.6688704.

[EGSVR16] Ittay Eyal, Adem Efe Gencer, Emin Gün Sirer, and Robbert Van Renesse. "Bitcoin-NG: A scalable blockchain protocol". In: *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 2016, pp. 45–59.

[ER60] Paul Erdös and Alfréd Rényi. "On the evolution of random graphs". In: *Publications of the Mathematical Institute of the Hungarian Academy of Science* 5.17-61 (1960), p. 43.

[ES14a] Ittay Eyal and Emin Gün Sirer. "Majority Is Not Enough: Bitcoin Mining Is Vulnerable". In: *Financial Cryptography and Data Security: 18th International Conference*. Springer, 2014, pp. 436–454. DOI: 10.1007/978-3-662-45472-5_28.

[ES14b] Ittay Eyal and Emin Gün Sirer. *How a mining monopoly can attack Bitcoin*. http://hackingdistributed.com/2014/06/16/how-a-mining-monopoly-can-attack-bitcoin/. June 2014.

[ES14c] Ittay Eyal and Emin Gün Sirer. *How to Disincentivize Large Bitcoin Mining Pools*. http://hackingdistributed.com/2014/06/18/how-to-disincentivize-large-bitcoin-mining-pools/. June 2014.

[Eya15] Ittay Eyal. "The Miner's Dilemma". In: *2015 IEEE Symposium on Security and Privacy*. May 2015, pp. 89–103. DOI: 10.1109/SP.2015.13.

[FBI12] Directorate of Intelligence. *Bitcoin Virtual Currency: Unique Features Present Distinct Challenges for Deterring Illicit Activity*. Tech. rep. Federal Bureau of Investigation, Apr. 2012.

[FIPS180-4] National Institute of Standards and Technology. *FIPS PUB 180-4: Secure Hash Standard (SHS)*. http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf. National Institute of Standards and Technology, Aug. 2015.

[FSY05] Amos Fiat, Jared Saia, and Maxwell Youg. "Making Chord Robust to Byzantine Attacks". In: *13th Annual European Symposium on Algorithms (ESA)*. Springer, Oct. 2005. DOI: 10.1007/11561071_71.

[GCKG14] Arthur Gervais, Srdjan Capkun, Ghassan O. Karame, and Damian Gruber. "On the Privacy Provisions of Bloom Filters in Lightweight Bitcoin Clients". In: *Proceedings of the 30th Annual Computer Security Applications Conference (ACSAC)*. ACM, 2014, pp. 326–335. DOI: 10.1145/2664243.2664267.

[Gil59] Edgar N. Gilbert. "Random Graphs". In: *The Annals of Mathematical Statistics* 30.4 (Dec. 1959), pp. 1141–1144. DOI: 10.1214/aoms/1177706098.

[GKCC14] Arthur Gervais, Ghassan O. Karame, Srdjan Capkun, and Vedran Capkun. "Is Bitcoin a Decentralized Currency?" In: *IEEE Security Privacy* 12.3 (May 2014), pp. 54–60. DOI: 10.1109/MSP.2014.49.

[GKL15] Juan Garay, Aggelos Kiayias, and Nikos Leonardos. "The Bitcoin Backbone Protocol: Analysis and Applications". In: *34th Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Advances in Cryptology - EUROCRYPT 2015. Springer, 2015, pp. 281–310. DOI: 10.1007/978-3-662-46803-6_10.

[GRKC15] Arthur Gervais, Hubert Ritzdorf, Ghassan O. Karame, and Srdjan Capkun. "Tampering with the Delivery of Blocks and Transactions in Bitcoin". In: *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2015, pp. 692–705. DOI: 10.1145/2810103.2813655.

[Hei14] Ethan Heilman. "One Weird Trick to Stop Selfish Miners: Fresh Bitcoins, A Solution for the Honest Miner". In: *Cryptology ePrint Archive* Report 2014/007 (2014).

[HKZG15] Ethan Heilman, Alison Kendler, Aviv Zohar, and Sharon Goldberg. "Eclipse Attacks on Bitcoin's Peer-to-Peer Network". In: *24th USENIX Security Symposium*. USENIX Association, Aug. 2015.

[Hun07] J. D. Hunter. "Matplotlib: A 2D graphics environment". In: *Computing In Science & Engineering* 9.3 (2007), pp. 90–95. DOI: 10.1109/MCSE.2007.55.

[JLGVM14] Benjamin Johnson, Aron Laszka, Jens Grossklags, Marie Vasek, and Tyler Moore. "Game-Theoretic Analysis of DDoS Attacks Against Bitcoin Mining Pools". In: *Financial Cryptography and Data Security: 18th International Conference*. Springer, 2014, pp. 72–86. DOI: 10.1007/978-3-662-44774-1_6.

[JMV01] Don Johnson, Alfred Menezes, and Scott Vanstone. "The Elliptic Curve Digital Signature Algorithm (ECDSA)". In: *International Journal of Information Security* 1.1 (2001), pp. 36–63. DOI: 10.1007/s102070100002.

[JOP+01] Eric Jones, Travis Oliphant, Pearu Peterson, et al. *SciPy: Open source scientific tools for Python*. [Accessed 2016-10-06]. 2001–. URL: http://www.scipy.org/.

[KAC12] Ghassan O. Karame, Elli Androulaki, and Srdjan Capkun. "Double-Spending Fast Payments in Bitcoin". In: *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS)*. ACM, 2012, pp. 906–917.

[KARGC15] Ghassan O. Karame, Elli Androulaki, Marc Roeschlin, Arthur Gervais, and Srdjan Capkun. "Misbehavior in Bitcoin: A Study of Double-Spending and Accountability". In: *ACM Transactions on Information and System Security (TISSEC)* 18.1 (May 2015), 2:1–2:32. DOI: 10.1145/2732196.

[KDF13] Joshua A Kroll, Ian C Davey, and Edward W Felten. "The Economics of Bitcoin Mining, or Bitcoin in the Presence of Adversaries". In: *The Twelfth Workshop on the Economics of Information Security (WEIS)* (June 2013).

[KJGK+16] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Nicolas Gailly, Ismail Khoffi, Linus Gasser, and Bryan Ford. "Enhancing Bitcoin Security and Performance with Strong Consistency via Collective Signing". In: *Proceedings of the USENIX Security Symposium*. 2016.

[KLS00] Stephen Kent, Charles Lynn, and Karen Seo. "Secure Border Gateway Protocol (S-BGP)". In: *IEEE Journal on Selected Areas in Communications (JSAC)* 18.4 (Apr. 2000). DOI: 10.1109/49.839934.

[LAL] Thibaut Lajoie-Mazenc, Emmanuelle Anceaume, and Romaric Ludinard. *TBD*.

[LNBZ+15] Loi Luu, Viswesh Narayanan, Kunal Baweja, Chaodong Zheng, Seth Gilbert, and Prateek Saxena. "SCP: a computationally-scalable Byzantine consensus protocol for blockchains". In: *Cryptology ePrint Archive* Report 2015/1168 (2015).

[Mer88] Ralph C. Merkle. "A Digital Signature Based on a Conventional Encryption Function". In: *Advances in Cryptology — CRYPTO '87: Proceedings*. Springer, 1988, pp. 369–378. DOI: 10.1007/3-540-48184-2_32.

[MGGR13] Ian Miers, Christina Garman, Matthew Green, and Aviel D. Rubin. "Zerocoin: Anonymous Distributed E-Cash from Bitcoin". In: *IEEE Symposium on Security and Privacy (SP)*. May 2013. DOI: 10.1109/SP.2013.34.

[MLPG+15] Andrew Miller, James Litton, Andrew Pachulski, Neal Gupta, Dave Levin, Neil Spring, and Bobby Bhattacharjee. *Discovering Bitcoin's public topology and influential nodes*. 2015. URL: `\url{https://cs.umd.edu/projects/coinscope/coinscope.pdf}`.

[Mor78] Jorge J Moré. "The Levenberg-Marquardt algorithm: implementation and theory". In: *Numerical analysis*. Springer, 1978, pp. 105–116. DOI: `10.1007/BFb0067700`.

[Nak08] Satoshi Nakamoto. "Bitcoin: A peer-to-peer electronic cash system". In: *White paper* (2008).

[NBFMG16] Arvind Narayanan, Joseph Bonneau, Edward Felten, Andrew Miller, and Steven Goldfeder. *Bitcoin and cryptocurrency technologies*. Accessed at `https://d28rh4a8wq0iu5.cloudfront.net/bitcointech/readings/princeton_bitcoin_book.pdf?a=1` on July 18th, 2016. Princeton University Pres, 2016.

[NG16] Cristopher Natoli and Vincent Gramoli. "The Blockchain Anomaly". In: *ArXiv* abs/1605.05438 (June 2016).

[OM14] Karl J. O'Dwyer and David Malone. "Bitcoin mining and its energy footprint". In: *Irish Signals Systems Conference 2014 and 2014 China-Ireland International Conference on Information and Communications Technologies (ISSC 2014/CIICT 2014). 25th IET*. June 2014, pp. 280–285. DOI: `10.1049/cp.2014.0699`.

[PS16] Rafael Pass and Elaine Shi. "FruitChains: A Fair Blockchain". In: *Cryptology ePrint Archive* Report 2016/916 (2016).

[PSS16] Rafael Pass, Lior Seeman, and Abhi Shelat. "Analysis of the Blockchain Protocol in Asynchronous Networks". In: *Cryptology ePrint Archive* Report 2016/454 (Sept. 2016).

[PVGM+11] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. "Scikit-learn: Machine learning in Python". In: *Journal of Machine Learning Research* 12.Oct (2011), pp. 2825–2830.

[Ref] *Bitcoin developer reference*. `https://bitcoin.org/en/developer-reference`. Accessed August 2nd, 2016.

[RFC4291] Robert Hinden and Stephen Deering. *IP Version 6 Addressing Architecture*. RFC 4291. RFC Editor, Feb. 2006. URL: `https://www.rfc-editor.org/rfc/rfc4291.txt`.

[Ros14] Meni Rosenfeld. "Analysis of Hashrate-Based Double Spending". In: *ArXiv* abs/1402.2009 (Feb. 2014).

[SZ15] Yonatan Sompolinsky and Aviv Zohar. "Secure High-Rate Transaction Processing in Bitcoin". In: *Financial Cryptography and Data Security: 19th International Conference*. Springer, 2015, pp. 507–527. DOI: 10.1007/978-3-662-47854-7_32.

[Tra14] Lawrence J Trautman. "Virtual currencies; Bitcoin & what now after Liberty Reserve, Silk Road, and Mt. Gox?" In: *Richmond Journal of Law and Technology* 20.13 (2014). http://jolt.richmond.edu/v20i4/article13.pdf.

[WCV11] Stéfan van der Walt, S. Chris Colbert, and Gaël Varoquaux. "The NumPy Array: A Structure for Efficient Numerical Computation". In: *Computing in Science & Engineering* 13.2 (2011), pp. 22–30.

Appendix A

Bitcoin data structures

Bitcoin defines several types of data structures. Some of them, such as the address manager, are never broadcast over the network and are thus entirely implementation-dependent. Others, however, constitute the basic building blocks of the system and its protocol, and thus follow standards. This appendix describes these structures. Whenever the structure has changed over time (such as blocks which have increased in size), the version described here is the latest recognized by v0.12.1 of the reference client [Core], protocol version 70 012.

An additional source of confusion is that, depending on the field, data is stored in big-endian (most significant bit at the leftmost position) or little-endian (most significant bit at the rightmost position). While of utmost importance when handling data, this information adds very little to the principles and is thus eluded.

A.1 Compact size unsigned integer

A compact size unsigned integer is an unsigned integer of variable length, used to decrease the memory and network requirements for counts that are often small but may need to be, occasionally, extremely large. It is as follows, where “below” means "less than or equal to", and 1^x is a short-hand for $\sum_{i=0}^{x-1} 2^i$:

1. Integers below `0xfc` (252) use one byte;
2. Else, integers below 1^{16} use two bytes prefixed by `0xfd` (253);
3. Else, integers below 1^{32} use four bytes prefixed by `0xfe` (254);
4. Else, integers below 1^{64} use eight bytes prefixed by `0xff` (255);

Bitcoin uses them when serializing vectors: they are always written as the count of entries (in compact size unsigned integer form) followed by all the entries, without any separation or ending token. In the remainder of this document, let cmpct(k) denote the length of the compact size unsigned integer representing the number k.

A.2 Coin

Bitcoin's monetary units are simply called bitcoins or coins. They comprise 10^8 *satoshis*, Bitcoin's smallest currency unit. Coins are not actually (digitally) represented by themselves in Bitcoin: they are only accessible in clusters, corresponding to transactions outputs. Anyone can trace the entire history of each satoshi back to the time it was minted through as part of the output of a coinbase transaction by following the chain of transactions spending it. However, this procedure requires some heuristics to compensate for their being completely fungible, such as considering that a given transaction transfers the coins by taking satoshis one by one, depleting successively each of its inputs to fill successively each of its outputs.

A.3 Transaction

This appendix describes what a transaction looks like, including their lock feature, the different statuses of transactions and finally how nodes verify their validity.

A.3.1 Transaction object

A transaction consists of two lists: the inputs (transactions that funded the accounts sending funds), and the outputs (accounts receiving funds). It is as follows:

1. An 8 B version number, currently set to 1;
2. The vector of inputs:
    a) A compact size unsigned integer count of entries;
    b) Then, each input is described as:
        i. the 32 B hash of the transaction used as an input;
        ii. a 4 B index indicating which of the transaction's outputs is used;
        iii. a compact size unsigned integer indicator of the length of the signature script;
        iv. a variable-length scriptSig (see Appendix B);
        v. a 4 B sequence number (see Appendix A.3.2);
3. The vector of outputs:
    a) A compact size unsigned integer count of entries;
    b) Then, each output consists of:
        i. An 8 B integer for the amount sent to that output. The value is given in satoshis (10^{-8} bitcoins);
        ii. a compact size unsigned integer indicator of the length of the payment script;

iii. a variable-length scriptPubKey (see Appendix B);

4. a 4 B lock time, see Appendix A.3.2.

A.3.2 Locks

Locking a transaction can have different meanings and wallets can do it in several ways: they can lock a whole transaction to make sure that no miners includes it in a block before a given event (called "lock event" in the following, and said "unlocked" when the condition it described is fulfilled), or lock any subset of its output to prevent anyone from spending them before almost arbitrary lock events. Those events can either be block heights or UNIX-like time stamps, and be described absolutely or relatively.

Absolute locks are defined by an unsigned integer, the lock time field. When less than $5 * 10^8$, it describes a block height; when greater than or equal to the same threshold, it describes a UNIX-like time stamp, that is a number of seconds elapsed since the UNIX epoch (00:00:00 on Thursday, January 1$^{\text{st}}$, 1970). Considering that the threshold correspond either to a number of blocks that would take on average 3.10^{11} seconds (approximately 9.5 millennia) to find or to some date in 1985, interpretation is unambiguous for the foreseeable future. However, it is only a UNIX-*like* time stamp because it is unsigned, contrarily to most UNIX implementations, and will overflow in 2106 rather than in 2038. The lock itself works as follows: a transaction is unlocked (and said *final*) if the lock event is strictly less than the current event (*i.e.* the time stamp or height of the block trying to include the transaction), or if all sequence numbers are equal to their maximum value, `0xffffffff` ($2^{32} - 1$).

Usually, a transaction is locked until the block on top of the highest one at the moment it was signed to avoid giving an incentive for miners to try and fork the blockchain; this is, however, in no way mandatory.

Relative lock, implemented in Core but not yet deployed as of August 19$^{\text{th}}$, 2016, is slightly different. First, it only applies to transaction whose version number is greater than or equal to 2. Each input defines a lock event if bit 31 of its sequence number is set to 0. It corresponds to a block height if bit 22 is set to 0 and to a time stamp otherwise. Bits 0 to 15 give the actual value of the lock, to be understood as "after the corresponding input was included in a block": thus, if the sequence number is equal to `0x80000001`, it means that the transaction cannot be included in the same block as the corresponding input (and, obviously, not before). Thus, a transaction can only be included when all its lock events are unlocked. An additional trick is that relative time stamps use a granularity of 512 seconds: `0x80400001` actually means that the transaction cannot be included in a block less than 512 seconds older than the block containing the corresponding input.

Wallets can also put absolute and relative locks in payment scripts (see Appendix B): one can ensure that a given output of a transaction cannot be spent before an almost arbitrary locking event, the main limitation still being the overflow of the spending transaction's lock time.

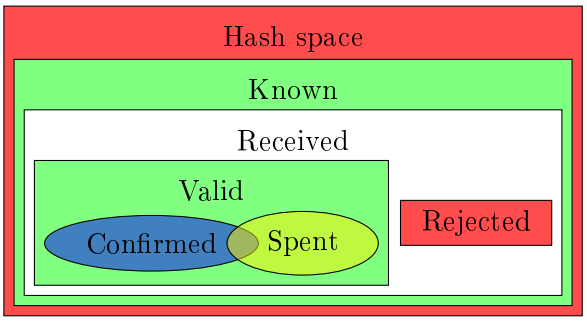

Figure A.1: Hierarchy of the statuses. The relative dimensions are not related to the relative sizes of the respective ensembles: the biggest one currently corresponds to unknown transactions.

A.3.3 Transaction status and type

Let Alice be a node and h a valid transaction hash (that is, currently, $h \in \{0,1\}^{256}$). Let $\mathcal{T}$ be the set of transactions whose hash is h. At any given time, Alice sees h in one of the following statuses:

1. unknown: Alice does not know any transaction in $\mathcal{T}$;
2. received: Alice knows one transaction from $\mathcal{T}$;
3. valid: Alice has checked that the transaction from $\mathcal{T}$ she received follows the rules, as per Appendix A.3.4;
4. confirmed: Alice's blockchain contains a block containing the transaction from $\mathcal{T}$ that she has received;
5. spent: another transaction received by Alice has used the one from $\mathcal{T}$ she knows about as an input;
6. rejected: the transaction from $\mathcal{T}$ Alice knows about has failed to pass the validity check.

These states are not all mutually exclusive: Figure A.1 shows their relative positions in the hash space. The evaluation is obviously dynamic: the most natural path is *unknown, received, valid, confirmed, and spent* and Core only keeps rejected transactions in memory for a short time before throwing them away; however, no transition is impossible, even though some are unlikely (from confirmed to unknown would most likely make a stop by received and rejected, which can happen in case of fork).

In addition to these context-dependent statuses, transactions have a type, which depend only on their content. Currently, Core defines three of those, which are mutually exclusive: standard, non-standard, and coinbase transaction. It considers transactions non-standard based on their scriptPubKey (see Appendix B), and a

transaction is said standard if it is neither non-standard nor a coinbase one (note that a transaction can be standard but invalid).

A coinbase transaction is the first transaction of its block and explains why most miners go through the effort of mining: it only takes one special input that does not refer to any past transaction, and still outputs some coins. Those have two sources: minting and transaction fees. Section 2.1.1 describes how to compute the amount for a given block.

As opposed to the other types, a coinbase transaction cannot be loose, i.e. it cannot be sent outside of its block (it cannot even be valid outside of it, since its output value depends on the other transactions included in the block). Moreover, Core protects coinbase transactions with a special lock: they can only be spent after a maturation period, currently set to 100 blocks. That is, if transaction T spends the coinbase transaction from block n' then miners can only include it in blocks at height n such that $n \geqslant n' + 100$.

A.3.4 Validity check

The core idea of Bitcoin is that anyone can verify any transaction: this is why the trusted third-party is superfluous. This means that transaction verification is of paramount importance. This section describes how Core [Core] performs it for loose transactions; it validates those received as part of a block during the block validation itself (see Appendix A.4.4). Other clients may run the tests in a different order but the result should be the same, as an invalid transaction invalidates any block that includes it, which could lead to hard forks.

Let Alice be a store receiving a loose transaction T.First, she goes through a check list of context-independent verifications:

1. T must have at least one input (coinbase transactions have exactly one input, even though it does not refer to a previous transaction);
2. T must have at least one output;
3. T must have a reasonable size (blocks must include a coinbase transaction so a transaction taking more than the maximum size of a block minus the minimum size of a transaction cannot be mined);
4. no output of T can be negative, and their total value cannot be greater than the total value of T's inputs;
5. all of T's input are distinct (i.e. T cannot perform a double-spend attack by itself);
6. all of T's inputs must *look* valid: the size of the signature script of coinbase transactions has lower and upper bounds, and all inputs of non-coinbase transactions must refer to a transaction. However, whether this reference is actually valid is only verified later on.

Then, Alice performs a series of context-dependent checks:

1. T must not be a coinbase transaction (which are, by nature, only confirmed or invalid);
2. T must have a valid version number (currently, Core only accepts 1; relative locks require version 2, which is not yet deployed);
3. T must be *final*: its absolute lock must already be unlocked (or disabled by the sequence numbers);
4. T must not be in conflict with any transaction it cannot replace: it can only use the same input as another transaction T' already in the mempool if all sequence numbers of T' are strictly less than `0xfffffffe` (used as a threshold instead of `0xffffffff` to allow the creation of locked non-replaceable transactions);
5. T must not already be in Alice's mempool;
6. Alice must have already received all of T's inputs (in case this check fails, Alice adds T to a list of orphan transactions rather than throwing it away);
7. no transaction has already spent any of T's inputs;
8. T's relative lock must already be unlocked;
9. T's input scripts must be standard (we give slightly more details in Appendix B);
10. T must have a reasonable number of script operators to make sure that it would not fill up a block by itself;
11. T must have a sufficiently high fee to have a chance of being mined;
12. Alice also rejects T if its fee is barely sufficient to accept it in her mempool and she has received a lot of those, using a counter multiplied by $(\frac{599}{600})^t$, where t is the time since she last received a small-fee transaction, comparing it to a threshold (by default, $1.5 * 10^4$), and incrementing it by the size of T;
13. T must not have too high a fee;
14. the set of T's unconfirmed ancestors (T's ancestors that are not yet in Alice's blockchain) must be reasonably small (no more than 25 elements for a total of up to 101 kilobytes), with the same constraints on the size of set of descendants of this set;
15. the sets of T's unconfirmed ancestors and of the transactions T would replace if accepted must not intersect;
16. it must be rational for Alice to mine T rather than all the transactions it replaces:

a) it must have a higher fee to size ratio than each of them;

b) not replace any transaction with too many descendants (which would require too much work to verify that T is better);

c) not have any unconfirmed ancestor that is not also an ancestor of at least one of the transactions T would replace;

d) it must have a total fee higher than the sum of what Alice would expect without replacement plus what she would drop in the replacement;

17. finally, Alice checks twice the inputs: first, performing all the standard checks and then, only the mandatory ones in case of bugs in some of the desired checks; standard but not mandatory checks include for example checking that the script does not contain any no-operation operators that may be redefined in the future. The procedure goes as follows for non-coinbase transactions:

    a) T's inputs must all be available (i.e. known to Alice and not yet spent);

    b) T's input values must all be in a valid range (i.e. not be negative or overflow);

    c) T's total output value must not be greater than its total input value;

    d) T's fee must be a valid amount (i.e. not be negative or overflow);

    e) all of T's scripts must return true (see Appendix B). Alice makes a difference between the desired and mandatory checks here: she skips the operators that are part of the standard but not mandatory checks in the second iteration.

When T has passed all of this, Alice removes the conflicting transactions, if there are any, from her mempool and adds T. If the mempool has grown bigger than 300 MB, she trims it down: she throws away all transactions older than 3 days and their descendants and, if necessary, the transactions with the lowest fee as well; this may include T itself.

A.3.5 Transaction graph

Figure A.2 shows a toy example of how the transactions form an acyclic directed graph which is the union of partially intersecting trees: the descendants of a given transaction form a tree whose leaves are UTxO and, similarly, the ancestors of a given transaction form a tree (when reversing its edges) whose leaves are coinbase transactions. It also highlights two common operations, which are to combine transaction outputs in a single transaction and, conversely, split a transaction in several outputs.

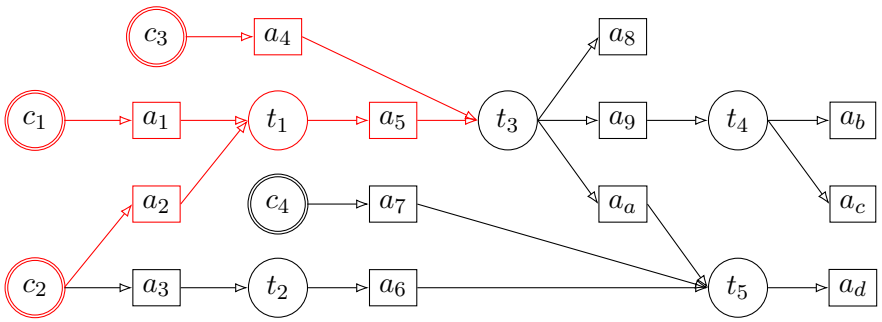

Figure A.2: Graph structure of the transactions. This example includes 4 coinbase transactions and 5 regular transactions. The rectangles correspond to accounts as defined by the formal model: the arrow pointing to one is the output creating it, and the arrow leaving from one is the input spending it. We show t_3's ancestors subgraph in red: its leaves are coinbase transactions.

A.4 Block

A block is a list of transaction: it provides *consensual sequential ordering* of *part* of *the events* that occurred in the Bitcoin network. In Bitcoin, events are just transactions, but the concept of blockchain needs not be that restricted. Miners may not include transactions in a given block for four reasons: some are invalid (or conflict with others, in which case only the one included in the main branch of the blockchain is valid, by definition), miners do not receive them before finding the block is found, miners may ignore some of them (e.g. those with an insufficient fee) and, finally, blocks have a size limit which cannot be exceeded. Sequential ordering is of paramount importance: one cannot spend funds before it receives them[1], and the only way to prevent money duplication when coins are not tied to a physical token is to make sure that only the first transaction sending a specific coin from Alice to anyone else is valid.

A.4.1 Block object

To hold all of these properties, a Bitcoin defines a block as follows:

1. A header, with the following fields:

    a) A 4 B Version field; Bitcoin Improvement Proposal (BIP) 9 [BIP9] changed it into a bitmask used by miners to vote on the acceptance of protocol modification. Thus, block 424 416, time stamped at 13:17:19 on August 9$^{\text{th}}$, 2016, has version number 536 870 912;

[1] A system of debt could work in Bitcoin as for any other currency: one needs to be at least lent some money before being able to spend it. The fact that it is seamless when using a credit card is barely a refinement of the system.

b) The 32 B hash of the block's parent in the blockchain;

c) The 32 B root of the block's Merkle tree (see Appendix A.4.3);

d) A 4 B POSIX time stamp, taken by the network with a grain of salt corresponding to the absence of clock synchronization;

e) A 4 B *base 256 scientific notation encoded* (see Appendix A.4.2) target. The hash of the header must be below that target;

f) A 4 B nonce, used as a space to search for a valid block hash.

2. A compact size unsigned integer count of transactions;

3. The transactions, in the same order as in the Merkle tree.

A block is valid if it passes a validity check described in Appendix A.4.4.

A.4.2 4-byte long base 256 scientific notation

Miners compact the target threshold in a 4-byte signed integer encoding representing a 32-byte unsigned integer. The expansion works as follows: the first byte is extracted as the exponent, and the last 3 bytes are the mantissa. It is shifted by exponent-3 bytes to the left, where a negative shift to the left corresponds to the opposite shift to the right and the -3 comes from the fact that the mantissa is already a 3-byte long number. As a safeguard, if the most significant bit of the mantissa is set (i.e. if the mantissa is negative in a signed integer representation) or the number is bigger than 2^{256} (which happens e.g. if the exponent is bigger than 34), it is replaced by 0. However, Core does not prevent underflow: it regards `0x020001ff` as a valid representation of 1.

The best compaction for a 256-bit long number a (the one introducing the smallest rounding error, used by the reference client) consists in shifting bytes so that a's most significant non-null bit ends up in its third least significant byte, adding as many leading or trailing zeroes as needed, keeping only the three least significant byte and prepending them with the appropriate exponent. The rounding error is the number represented by the bytes that are shifted out. Figure A.3 illustrates compaction and extension.

Let us denote by f the function that expands a 4-byte number into a 32-byte one, and f^{-1} that which performs the compaction.

Definition 20
$a \in \{0,1\}^{256}$ *is said to have an* exact encoding *if and only if* $f(f^{-1}(a) = f^{-1}(f(a)) = a$*, that is if it is possible to represent* a *in 4-byte long base 256 scientific notation without rounding it.*

Definition 21
$a \in \{0,1\}^{32}$ *is said to be a* licit encoding *if and only if its expansion does not overflow and the highest bit of its mantissa is set to 0. Conversely, it is said to be* illicit *if it*

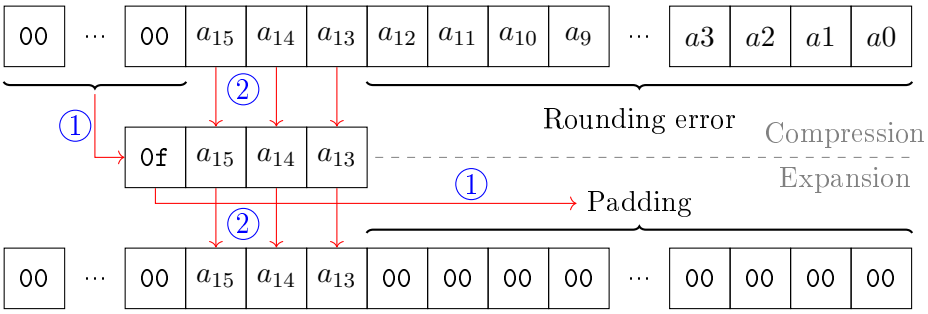

Figure A.3: From 256 bits to 32 and back again: the 4-byte long base 256 scientific notation. The gray line separates compression (upper half) from extension (lower half). Compression consists in storing the 31-complement of the number of leading zero bytes as the most significant byte of the compact form and then copying the next three bytes. Extension consists in copying the three least significant bytes into a position specified by the most significant one.

is not licit, and their compaction is 0 by convention. We denote by $\mathcal{S}$ the set of licit encodings.

Let us also index bits by increasing significance: the least significant bit is bit 0, and the most significant bit is 255 or 31 depending on whether the number is a 256 or 32-bit long one. It is obvious that f is not surjective: its domain is finite and smaller than its codomain. Its image is actually smaller than 32 bit: all numbers with bits 23, 30, or 31 set are illicit and have the same image. This gives a 29-bit upper bound, which is strict since two different numbers can have the same expansion (e.g. `0x04000001` and `0x03000100` which are both equal to 256). Property 6 gives the exact size of the image of f.

Property 6
The transformation from 4 to 32-byte long integers using the base 256 scientific notation has an image of size $2^{29} - 2^{28} - 2^{21} - 2^{16} - 2^9 + 5 = 266,272,261 \approx 0.496 * 2^{29} \approx 2^{28}$.

Proof. First, illicit numbers compacted to 0 include all those with bit 31, 30, or 23 set: this leaves only 2^{29} numbers. They also include the $29 * 2^{23}$ numbers with exponent between 35 and 63 (both included), the $(2^7 - 1)2^{16}$ with exponent equal to 34 and at least one bit set among the 7 least significant bits of the most significant byte of the mantissa and the $(2^{15} - 1)2^8$ with exponent equal to 33 and at least one bit set among the two most significant bytes of the mantissa except bit 23, all of those for overflow reasons (their most significant bit would have an index greater than 256 after transformation).

Let $a, b \in \mathcal{S}, b_3 \leqslant a_3, b \neq a$, where x_i is the i-th byte of x using the same indexing convention as for bits (thus, the exponent is byte 3).

We draw the following result from the uniqueness of the decomposition of a number in base 256, where $(x_2x_1x_0) = x_2256^2 + x_1256 + x_0$:

$$f(a) = f(b) \Leftrightarrow (a_2a_1a_0) * 256^{a_3-b_3} = (b_2b_1b_0)$$
$$\Leftrightarrow \begin{cases} a_i = 0 & \forall i, 3-(a_3-b_3) \leqslant i \leqslant 2 \\ b_j = 0 & \forall j, 0 \leqslant j \leqslant a_3 - b_3 - 1 \\ a_i = b_{i+a_3-b_3} & \forall i, 0 \leqslant i \leqslant 3-(a_3-b_3) \end{cases}$$

We get $a_3 - b_3 = 0 \Leftrightarrow (f(a) = f(b) \Leftrightarrow a = b)$. Hence, in the following, we replace $b_3 \leqslant a_3, b \neq a$ by the equivalent $b_3 < a_3, b \neq a$.

Thus, $a_2 \neq 0 \Rightarrow \nexists\, b \in \mathcal{S}, b_3 < a_3, b \neq a, f(a) = f(b)$.

Let $a_2 = 0, a_1 \neq 0$. Then we have $a_3 - b_3 = 1$ and thus if $a_3 > 0, !\exists b$ that meets our requirements, that is $b = (a_3 - 1)256^3 + a_1256^2 + a_0256$, which indeed satisfies $b \in \mathcal{S}$. This decreases the size of f's image by $2^8(2^8 - 1)33$: for each of the 33 licit exponents (1 to 33, both included since no bit is set in a_2 but one is set in a_1), a_0 can be anything and a_1 can be anything but 0.

Let now $a_2 = a_1 = 0 \neq a_0$. Then we have $a_3 - b_3 \in \{1, 2\}$. The b's such that $f(a) = f(b)$ are either $(a_3 - 1)256^3 + a_0256$ or $(a_3 - 2)256^3 + a_0256^2$, valid only for $a_3 \geqslant 1$ and 2 respectively. However, we have already counted both of those when $a_3 \geqslant 2$: the former is one of the a's of the previous case, and the latter is the corresponding b. We thus need to count the a's of this group, and the b's only when $a_3 = 1$ (because this case did not arise in the previous case): there are respectively $34(2^8 - 1)$ and $2^8 - 1$ of them.

Finally, we exclude the licit representations of 0 as well: they comprise numbers with all bits of the mantissa set to 0 (34 choices of exponent), with exponent equal to 2 ($2^8 - 1$ choices of least significant byte, the other two are equal to 0), equal to 1 ($2^{16} - 1$ choices for the two least significant bytes, the other one is equal to 0), or equal to 0 ($2^{23} - 1$ choices for the mantissa, given that its highest bit is always zero), where the -1's come from the fact that we have already counted a mantissa equal to zero. We have to include one of them back, though, because otherwise we would have excluded all pre-images of 0.

We derive the result by summing all of these terms, expanding them and using $32 = 2^5$.

□

What this means is that this encoding sacrifices seven eighth of its size to gain the ability to store numbers with a very large amplitude (from 0 to $\sum_{i=0}^{22} 2^{232+i} = 2^{255} - 2^{232}$). The numbers that have exact encodings all share a similar form: the index difference between their most significant non-null bit and the least significant (possibly null) bit of their least significant non-null byte is at most 22. The equality case corresponds to those number that have exactly one exact encoding; the numbers falling in the strict inequality case have different equivalent exact encodings.

This happens because of the weak definition of 4-byte-long base 256 scientific notation: contrarily to the usual scientific notation, there is no constraint for the most significant byte of the mantissa not to be zero, even though the reference client tries to apply that logic. However, it would be pointless to enforce such a rule: it would either require more work from the stores to validate block headers (as negligible as that amount of work would actually be), or that they trust the network input to be correct, while the only sane assumption is that any input is incorrect or malicious until proven otherwise. Furthermore, stores reject block with a target different from the one they expect: only the decoded 256-bit-long target actually matters to them.

Finally, the rounding error is manageable: let $a_i = 2^i + \sum_{j=0}^{j=i-17} 2^j \ \forall i \in 8\mathbb{N}, i \leqslant 255$, that is the number with the least significant bit of byte $i/8$ and all the bits from all the bytes of index at most $i/8 - 3$ set, and all the other bits equal to 0. In this case, the relative rounding error Δ_{f^{-1},a_i} is:

$$\begin{aligned}
\Delta_{f^{-1},a_i} &= \frac{a_i - f(f^{-1}(a_i))}{a_i} \\
&= \frac{\sum_{j=0}^{j=i-17} 2^j}{2^i + \sum_{j=0}^{j=i-17} 2^j} \\
\Delta_{f^{-1},a_i} &= \begin{cases} 0 & \text{if } i \leqslant 16 \\ \frac{2^{i-16}-1}{2^i(1+2^{-16})-1} & \text{if } i \geqslant 16 \end{cases}
\end{aligned}$$

It is clear that a_i maximizes the relative rounding error over $\{0,1\}^{8+i}$: the best mantissa is as small as possible while the number of bits that are shifted out is as large as possible[2]. It is monotonic and increases with i, with a limit of $\frac{1}{2^{16}+1} = \frac{1}{65537} \approx 1.5 * 10^{-5}$ for $i \to \infty$[3]. This means that even though rounding down the target makes it harder for miners to find blocks, the effect is barely noticeable (and affects all miners the same way, which is an essential fairness requirement).

A.4.3 Merkle tree

Bitcoin uses Merkle trees [Mer88] to allow verification of the entire list of transactions included in a block using only 256 bits in the block's header. The Merkle tree of a list of n transactions is an unbalanced binary tree with n leaves at its deepest level, $k_n = \lceil \log_2(n) \rceil$, which are the hashes of the transactions. Algorithm 1 is a pseudocode description of how to build the entire tree; basically each node is the double SHA-256 hash of the concatenation of its two children.

Bitcoin uses this structure because it is very efficient. Thus, in order to verify that a given transaction of interest is one of the tree's leaves, one only needs a

[2]Given that the domain of f^{-1} is finite, it could be rigorously proven by computing the relative rounding error for each possible input.

[3]This limit does not actually make sense in our setting because i is upper-bounded by 248 but it is good enough an upper bound to get the idea.

Algorithm 1 How to build a Merkle tree from an ordered set of transactions. h is the double SHA-256 hash function and $||$ is the concatenation operator.

1: **procedure** BUILD($t_0, ...t_{n-1}$) ▷ Build tree containing transactions t_0...
2: $maxDepth \leftarrow \lceil \log_2(n+1) \rceil$
3: **for** $i \leftarrow 0, .., n-1$ **do** ▷ Initialise the leaves
4: Create leaf $h_{maxDepth,i} = h(t_i)$
5: **for** $k \leftarrow maxDepth - 1, ...0$ **do** ▷ For each level
6: **for** $i \leftarrow 0, ..., 2^k$ **do** ▷ For each node
7: **if** $h_{k+1,2i}, h_{k+1,2i+1}$ have been defined **then**
8: Create node $h_{k,i}$ with value $h(h_{k+1,2i}||h_{k+1,2i+1})$ and children $h_{k+1,2i}$ and $h_{k+1,2i+1}$
9: **else if** $h_{k+1,2i}$ has been defined **then** ▷ Unbalanced tree with uneven number of nodes.
10: Create node $h_{k,i}$ with value $h(h_{k+1,2i}||h_{k+1,2i})$ and child $h_{k+1,2i}$
11: **break**
12: **else** ▷ Unbalanced tree with even number of nodes.
13: **break**
14: **return** the generated nodes

number of elements that is logarithmic in the number of transactions included in the tree: the two children of each node on the path from the transaction to the root.

However, the receiver of such a message needs to know where those hashes belong in the tree, otherwise the transmission is meaningless. Bitcoin uses bit flags for this, using a depth-first traversal of the tree: this increases the size of the transmission by $\lceil \frac{h}{8} \rceil + \text{cmpct}(\lceil \frac{h}{8} \rceil)$ bytes, where h is the number of transmitted hashes. When the sender provides a single hash (which happens only when a block only contains a coinbase transaction), this only increases the size of the transmission by a factor of $\frac{1}{16}$ (2 bytes for a 32-byte hash), and it decreases even further when there are more hashes involved.

The flag bits are a boolean evaluation of the question "does the subtree rooted at this node needs further exploration?". Algorithm 2 is a pseudocode summary of how the reference client[4] produces the lists of flags and hashes needed to verify that a Merkle tree of given root contains a certain set of transactions, while Algorithm 3 summarizes how to use a list of hashes and flags to verify that a tree has a given root. From that point, one only needs to verify that the transactions of interest appear at the right place in the list of hashes to be sure that the tree contains them. Figure A.4 displays an example of Merkle tree for a block with 7 transaction and highlights the hashes needed by someone who would like to verify the presence of transaction t_3. It shows which four of them suffice to recompute the root (included in the block's header) while making sure that it contains t_3's hash, assuming SHA-256's collision

[4] As it sends these structures over the network, any other procedure *must* give the exact same result to be valid in this context.

resistance.

Algorithm 2 Depth-first traversal of a Merkle tree to determine which hashes are needed to verify that the tree contains a set of transactions; the pseudocode takes an object-oriented approach, assuming that the nodes of the tree are accessible through their position described as (depth, index at depth).

```
1: procedure CompactRecurse((i, j), t_0, ...t_p, flags, hashes)   ▷ Determine flag corresponding to node i, j and whether its value is necessary or redundant
2:     if the subtree rooted by node (i, j) contains at least one of t_0, ...t_p then
3:         Append bit set to 1 to flags
4:     else                                                       ▷ The subtree is not useful here.
5:         Append bit set to 0 to flags
6:     if node (i, j) is a leaf or the subtree that it roots does not contain any of t_0, ..., t_p then
7:         Append its value to hashes and return
8:     else                ▷ Recurse through the subtree to find exact location of useful information
9:         CompactRecurse((i + 1, 2j), t_0, ...t_p, flags, hashes)
10:        if node (i + 1, 2j + 1) is defined then
11:            CompactRecurse((i + 1, 2j + 1), t_0, ...t_p, flags, hashes)
12:    return
```

However, Merkle trees are vulnerable to duplication attacks: if the number of transactions is not a power of 2, then the binary tree is unbalanced and some intermediate nodes are computed as the double hash of the concatenation of its only leaf with itself. An attacker can leverage this to include the transactions such that this node gets two identical children; this leaves its hash, and thus everything above it, unchanged even though the deepest level of the tree is different. Bitcoin shields itself against this by considering as invalid any tree containing a duplicate hash.

A.4.4 Validity check

Just as transactions, blocks are verifiable by the network. Let Alice be a store, b a block she has just received and $t_0, ...t_n$ its transactions. In this section, we use "below" to mean "less than or equal to".

First, she goes through a check-list of context-independent verifications:

Algorithm 3 Depth-first traversal of a Merkle tree to verify that it includes the input hashes at positions specified by the input flags; the pseudocode takes an object-oriented approach, assuming that the nodes of the tree are accessible through their position described as (depth, index at depth).

```
1:  procedure VERIFY(count, hashes, flags, root)        ▷ Verify that the tree
    made of count transactions with specified hashes at position specified by flags
    has the given root.
2:      Build empty Merkle tree with count leaves
3:      Verify that root = VERIFYRECURSE((0, 0), flags, hashes)
4:  procedure VERIFYRECURSE((i, j), flags, hashes)      ▷ Compute hash of node
    (i, j)
5:      f ← pop(flags)      ▷ pop removes and returns the first element of its input.
6:      if f = 0 or node (i, j) is a leaf then
7:          return pop(hashes)
8:      else
9:          left ← VERIFYRECURSE((i + 1, 2j), flags, hashes)
10:         if node (i + 1, 2j + 1) is defined then
11:             right ← VERIFYRECURSE((i + 1, 2j + 1), flags, hashes)
12:         else
13:             right ← left
14:         return h(left||right)          ▷ h is the double SHA-256 hash function
```

1. She verifies its header:
   a) b's PoW must be valid: the target must be below its upper bound (set to $2^{224} - 1$ on the main network) and the block hash must be below the target;
   b) b's time stamp must be less than 2 hours younger than the current network time, computed using the median offset of some of Alice's neighbours;
2. She verifies its Merkle root and that the Merkle tree does not include duplicated transactions
3. b's serialization must be at most 1MB and it must contain at least one transaction;
4. b's first transaction must be a coinbase transaction and none of the others can;
5. each of b's transactions must pass the context-independent transaction validity check (see Appendix A.3.4);
6. b's transactions must not contain more than a total of 20 000 script signature operations;

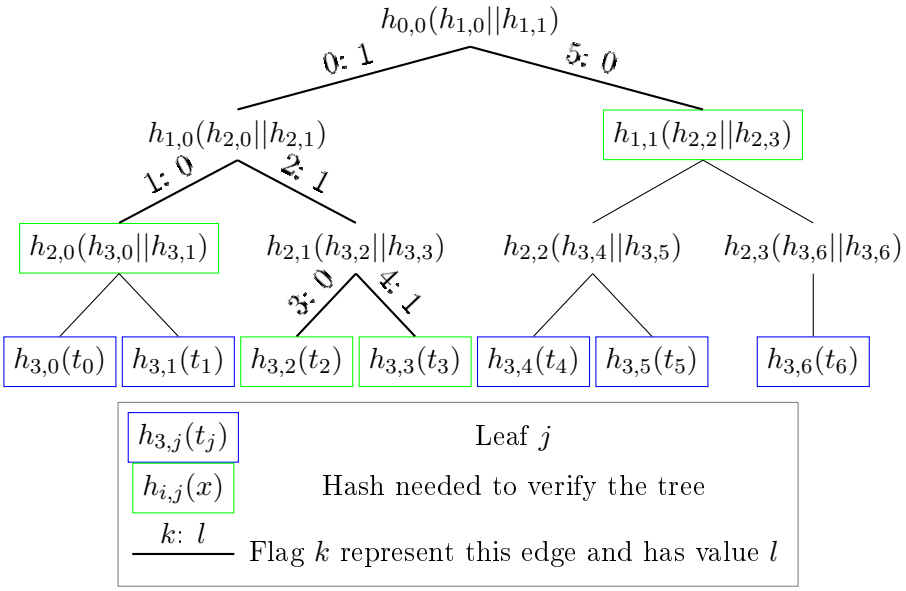

Figure A.4: Example of Merkle tree for a list of 7 transactions, where $h_{i,j}(\cdot)$ is a shorthand for $h_{i,j} = h(\cdot)$ and $h(\cdot)$ is the double SHA-256 function. In this example, the sender would send $h_{2,0}, h_{3,2}, h_{3,3}, h_{1,1}$, the bits `101010` (padded with 3 trailing zeroes to occupy an integer number of bytes) and t_3 along with the block header to let the receiver verify that the block contains t_3.

Then, she checks if she should accept the block, which includes the following checks:

1. she checks the validity of b's:
   a) b's hash is not that of a known invalid block;
   b) b's header is valid (exactly as in item 1 of the context-independent checks);
   c) Alice knows $p(b)$ and considers it valid;
   d) b is not trying to get inserted in the blockchain before the last checkpoint (see Appendix A.5);
   e) b passes the contextual header checks:
      i. b's target must be exactly what Alice expects
      ii. b's time stamp must be ulterior to the median of the previous 11 block time stamps
      iii. b's version must be strictly greater than 4
2. Alice must either have requested b or it must be better than her current tip and not be more than 288 blocks higher in the blockchain. In case Alice has previously received it, she only goes on if she requested it (e.g. because she has pruned it);

3. b must pass (again) the context-independent validity checks;
4. b must pass the contextual validity checks:
    a) all transactions must be final (see Appendix A.3.2);
    b) the coinbase transaction must start with the block height

At this point, if everything went fine, Alice has accepted and stored the block and only need to change the tip of her blockchain, if appropriate. Before adding new blocks, she first removes all those that are no longer in the main branch (which happens only if the previously loosing branch of a fork just (temporarily) won the race and became the longest one). Then, for each block newly accepted in the main branch (that we will all denote b for a sake of simplicity and because the most common scenario is that the newly received block was not involved in a fork), she goes through the following check-list:

1. b must pass the context-independent validity check;
2. b's parent must be the current tip;
3. b must not contain any transaction with the same hash as one previously included in the blockchain[5];
4. b's transactions must not contain more than a total of 20 000 script signature operations;
5. All of b's transactions' inputs must be available (i.e. known and unspent);
6. When relative locks (see Appendix A.3.2) are active, they must be unlocked for each of b's transactions;
7. All of b's non-coinbase transactions' must pass the input-related check list described as item 17 of the context-dependent transaction validity check[6] (see Appendix A.3.4);
8. b's coinbase transaction must have an output value below the block reward (fees included);

After each successful completion of the check list, Core throws away the in-mempool transactions conflicting with the block (consuming an input already consumed by another transaction it contains) and updates a lot of housekeeping variables.

When the iteration is over, Alice has finished validating data and can switch back to networking, as detailed in Appendix C.

[5]There are two exceptions to this rule: blocks at height 91 842 and 91 880 each duplicated a previous coinbase transaction, which prevents the spending of the first instance.

[6]With a twist: script verification is actually performed in the background by another thread, joined after the next item.

A.4.5 Initial block download

Initial block download is an operation performed by nodes at initialisation to catch up on the state of the blockchain. It happens either when the node first joins the network with an empty database, when its blockchain tip is more than 24-hour old or when the tip of its blockchain is more than 144 blocks (24 hours worth of blocks) behind its chain of headers.

Let Alice be a node initialising her blockchain, with Bob and Carol as neighbours. She starts by picking one of them: if possible, an outbound connection to a full node, but if no such connection exists, an inbound one to a full node suffices. She then sends this selected neighbour a `getheaders` message requesting for all headers after (and including) Alice's current best one. As soon as she gets the response, she can run three operations in parallel: keeping asking for more headers until the response is not full (that is, contains less than 2000 headers), indicating that she has reached her neighbour's tip, validating the headers to initialise the chain of headers, and request the blocks corresponding to the headers she has validated.

She only asks one neighbour for all headers until the best header she has is less than 24-hour old, at which point she asks all of her neighbours to confirm that the one she downloaded from was up to date and not feeding her an illegitimate chain. However, since blocks are much heavier than headers, she distributes the `block` requests between all of her neighbours to avoid being slowed down by a neighbour's upload speed or to saturate their upload quotas.

Finally, if she detects that a neighbour supposedly feeding her headers or blocks takes too much time, she drops the connection to try and find a more efficient node in the network.

A.5 Blockchain

Before anything else, the blockchain is a database: each store uses it to maintain a record of all the transactions accepted by the system in the form of blocks to ensure that they all agree on their sequential ordering, and most of Bitcoin's operations manage it locally (query for or insert data and ensure integrity) or distribute it (exchange data with neighbours). As any huge database management system, it resides partly on the store's hard drive and partly in main memory; the way Bitcoin Core (or any other client) performs indexing, information retrieval and long-term storage is both complex and completely out of the scope of this work.

The structure of chain derives from blocks linking to their parent: from any block, one can iteratively follow all the "previous block" pointers until reaching a block without one, which is the starting point of the chain including the starting block. However, the other direction shows a different situation: nothing (structurally) prevents a block from having two (or more) children, leading to an actual tree structure. This is the fork phenomenon. When this happens, each rational blockchain store defines its main branch as the heaviest path it knows in the blockchain, where the weight of a path is the sum of the difficulties of all the blocks it

includes. The genesis block `000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f` roots all valid paths, which can only go from a block to one of its children and not the other way around. Using the weight instead of the length of a branch prevents attackers from preparing a very long chain of easy blocks and feeding it to the network.

Since nodes only accept transactions compatible with the history described by their main branch, forks are solved when a branch becomes heavier than its competitors and all nodes accept it as their main one; after some time, necessary to be reasonably sure that no miner works on extending a losing branch, nodes can prune it from the tree that is the database. Another attack could be to work on a very early fork (say, diverging from the main branch right after the genesis block) without feeding it to the network before it gets longer than the main branch. To prevent this, Bitcoin Core used to hard-code some checkpoints in the blockchain: each Core store knows the that the commonly accepted block at height 295 000 has hash `000000000000000004d9b4ef50f0f9d686fd69db2e03af35a100370c64632a983` and that its time stamp corresponds to Wednesday, April 9$^{\text{th}}$, 2014 at 21:47:44 GMT. However, it is also the last checkpoint: Core dropped the system because the community considered it a threat to Bitcoin's decentralisation and assumes that it would be infinitely more profitable for an attacker that could pull off this attack with non-negligible probability to simply devote his computing power to mining on the main branch.

A.6 Bitcoin address

Bitcoin addresses are a way to encode public key hashes that is more secure by including error detection codes allowing a user to be sure that she is not sending funds somewhere else than where she thinks.

To do this in a way that is compact but not error prone, Bitcoin defines the base 58 encoding, whose characters are numbers from 1 to 9 and lower and upper case letters except for "O", "I", and "l"; that is, base 62 (regular English characters) without the two pairs of characters that may be hard or even impossible to distinguish when printed in certain fonts.

An address is computed as follows: first, the public key is hashed to 160 bits (using the regular SHA-256, RIPEMD-160 double hash). Then, the suffix 0 is added, and the whole message of 21 bytes is double SHA-256 hashed. The first 4 bytes of this second hash are added as a suffix, to be used as a checksum. Finally, the 25 bytes are converted to base 58 encoding.

In the process, the leading zeroes are treated in a peculiar way: they are counted and skipped. After the conversion of the rest of the string, as many "1" as there were zeroes are prefixed, which explains why all Bitcoin addresses start with "1".

One can also encode the address of a script instead of a key, for a pay-to-script-hash scriptPubKey. The procedure is the same, except that the suffix added is 5, converted to 3 in base58.

A.7 Network address

Bitcoin addresses refer to users, but Bitcoin routers need to find each other over the Internet to form the network. Very naturally, they use network addresses, each of which mostly consists of an IP address and port number pair. Bitcoin supports 128-bit long IPv6 addresses, and to avoid having to handle two different lengths IPv4 addresses are mapped to IPv6 ones of the form `::FFFF:0:0/96` (that is, 80 zeroes, 16 ones, and the 32 bits of the IPv4 address) as per RFC 4291 [RFC4291]. Tor [DMS04] is also supported.

However, Bitcoin also adds two less common pieces of information to network addresses: a bitmask of the services supposedly provided by the peer (see Appendix D.3.1) and a time stamp that each node updates based on very peculiar rules from the very moment it discovers the address.

To describe them, let Alice be a node who hast just received an `addr` message from a neighbour. In all this section, Alice's clock (and time) actually refer to her *adjusted* time: she modifies the UNIX time stamp of the device she runs on by an offset computed using some of the `version` handshakes she has performed. After the usual sanity checks, she decides to store the addresses in her address manager. For each address a, she checks the associated time stamp that Bob included in the message. If it is prior to 10^8 (Saturday, March 3$^{\text{th}}$, 1973 at 09:46:40 GMT) or if it is more than 10 minutes in the future for Alice's clock, she replaces by her current time aged by one day. If the address is reachable, she adds it to her address manager; in the process, she ages the time stamp by 2 hours.

The address manager defines the update period of the address' time stamp to be either one hour is the time stamp is less than 24-hour old when added, or 24 hours otherwise: Alice will only update the time stamp when she receives again the address with a greater time stamp than the one she has plus this update period and the 2-hour penalty. In other words, she only updates the time stamp if it makes it move by more than the update period. As usual, there is a catch: if Alice manages to establish a connection with that address (no matter its direction), she updates its time stamp to her current time if the time stamp is more than twenty minutes old.

When advertising addresses (unsolicited or in response to a `qetaddr` query), Alice includes the time stamp she has for each of them; there is no final trick.

A.8 Bloom filter

A Bloom filter [Blo70] is a regular computer science probabilistic data structure used to check if an element is part of a set. There is a bias: it may generate false positives but no false negative: it answers "definitely not" or "probably yes". The rate of false positive depends on the size of the filter and the number of elements in the set.

Bitcoin uses them specifically for determining if transactions are of interest to SPV nodes, with two goals in mind: making sure that SPV nodes use as little

resources as possible while somewhat preserving their privacy.

Sending only transactions of interest to SPV nodes helps them avoid consuming resources to receive and validate "useless" data. However, at the same time it gives away all of their Bitcoin addresses to their neighbours, which endangers Bitcoin's pseudonymity. Thus, a structure that never underestimates the importance of a piece of information but sometimes overestimates it is particularly well suited, especially if it is space and time efficient, which the Bloom filter is.

However, given that wallets should not reuse addresses, filters need to be updated every time they find a match to include the new address of interest. The `filterload` message configures how and who performs this. It can be left to the full node to update the filter to match against all outputs of every matching transaction: this will slowly but steadily make the rate of false positive grow (because the filter has a fixed size) but will not leak any more information than loading the initial filter. He can also never update it and wait for the SPV node to send him `filteradd` messages, which definitely leaks information as the elements to add to the filter are not obfuscated (and still slowly increases the rate of false positive, though at a slower pace). Finally, the SPV node can also send an entirely new filter through a `filterload` message (which does not need to follow a `filterclear` one) every time it needs to be updated. The main downside of this approach is that it consumes more resources to recompute and resend the filter every time it needs to be updated.

Appendix B

scriptPubKey and scriptSig

Bitcoin uses a non-Turing complete scripting language[1] simply called Script to determine whether a transaction is actually allowed to use its inputs. It is stack based and a script is valid if it can go through its entire execution without failure and leave the stack with a top value different from (negative) zero. A transaction is valid if all of its input scripts are valid. However, Core considers certain scripts non-standard and does not relay them outside of a block (thus, it is up to the emitter to make sure it reaches miners) even if they can be valid with the right signature script. As described in Appendix A.3.1, transactions include two types of scripts: each output contains a scriptPubKey and each input a scriptSig. However, both kinds use the same language: the difference is only made to separate the parts provided by the emitter and the redeemer of a given coin.

Rather than going through the 256 *opcodes* (Script commands) and produce an extensive guide to Script, this appendix gives first a general idea of the kinds of operations that Script defines, followed by an example of how a specific transaction was spent (and how said transaction did not follow some of Bitcoin's security recommendations). Most of its content is adapted from the Script page of the Bitcoin Wiki [BW] and from the source code of version 0.12.1 of the reference client, specifically `src/script/script.h`

The opcodes can be grouped in several categories based on their action on the stack:

Value-pushers Push a variable number of items on the stack;

Branching conditions Classic `if then (else) endif` structure, along with ways to mark a transaction as invalid;

Stack operators Modify the stack by duplicating, erasing, or moving data around;

Splice operators Most of these are disabled, which make the script fail if present;

[1] It purposely does not contain loops. Ethereum [But14] is an example of altcoin that changed this.

Bitwise operators Most of these are disabled as well, those left can however check equality between two items;

Arithmetic operators Perform several arithmetic operations such as additions or comparisons;

Cryptography Perform cryptographic operations such as computing hashes or verifying signatures;

Expansions Ten No-operation words have been defined, out of which two have been redefined to check the validity of the transaction lock time.

Using carefully selected operators in those categories, one can enforce that only the intended recipient of a transaction can use its output as an input for a new one. The most usual way to do this is through the pay-to-pubkey-hash, which we describe in the first part of the following example, examining the two outputs of transaction `3a6ffefa2be34b63ebcdadfeadb4d2cb3a76f625f35d38907b6b8355dccce874` (`3a6f` in the following), from block 424 151. Its two outputs contain the following scriptPubKey, where we add white spaces to separate the different components of the scripts:

1. `OP_DUP OP_HASH160 c6b3edff7379d3f58146e457110f1c4ab7d50eb6 OP_EQUALVERIFY OP_CHECKSIG`

2. `OP_HASH160 6883100c446c4652bf40030166c25bf432f75ceb OP_EQUAL`

We denote by `hash1` and `hash2`the two hashes in the following.

Its first output was spent as the first input of transaction `b6c79b3935697b9fdbb6b88040442a1b669c73b91339661d0bbdb23721853c42`, from block 424 238. The corresponding ScriptSig (script included in the spending transaction to collect and use the funds) comprises two components: first, a signature (`304402205ac8a31667895f36bf23f85e518b4f6102f33555d8ed15e421d08d35d679d08302204c7af33bebd58f0539bbf831ab69281e3f17e9cc7c1e5caa21ee4dda773d6eb701`), denoted `sig` in the following, followed by the corresponding compressed public-key (`03d42179014ca72d9f0c1ee8dda35dd3ccde55d2a66cf403afcee9b6c66ddfd656`), denoted `pubkey` in the following. The script to execute is the concatenation of the input script and the output one, giving `sig pubkey OP_DUP OP_HASH160 hash1 OP_EQUALVERIFY OP_CHECKSIG`, where the value-pushing opcodes have been omitted before each of the three constants. Table B.1a shows its execution.

The operation performed when reading `OP_EQUALVERIFY` verifies that the public key provided by the spender corresponds to the address the input refers to by deriving the latter from the former. Then, `OP_CHECKSIG` uses the public key to verify the ECDSA signature.

The second output was spent as the only input of transaction `6539831e3fd1794f5a0e56ea7bc5cada9c39325c6f62bdd93907a8e7103e68bd`, from the same block. The corresponding scriptSig contained two elements: `OP_FALSE` and 215 B of data

Read from script	Operation performed	Resulting stack
`sig pubkey`	Push constants on stack	`sig pubkey`
`OP_DUP`	Duplicate top stack item	`sig pubkey pubkey`
`OP_HASH160`	Hash using SHA-256 and then RIPEMD-160 top stack item	`sig pubkey hash(pubkey)`
`hash1`	Push constant on stack	`sig pubkey hash(pubkey) hash1`
`OP_EQUALVERIFY`	Consume and compare top two stack items. If different, fail the script	`sig pubkey`
`OP_CHECKSIG`	Check if second-to-top stack item is the signature of the transaction made by top stack item. Push result of the check on stack	`1`
`<empty>`	If top stack item is not `0`, script is valid. Otherwise, fail	*Script has returned*

(a) Execution of the script spending the first output of `3a6f`.

Read from script	Operation performed	Resulting stack
`OP_FALSE data`	Push constants on stack	`0 data`
`OP_HASH160`	Hash top stack item to 160 bit	`0 hash(data)`
`hash2`	Push constant on stack	`0 hash(data) hash2`
`OP_EQUAL`	Test equality of the top two stack items and push the result on stack	`0 1`
`<empty>`	If top stack item is not `0`, script is valid. Otherwise, fail	*Script has returned*

(b) Execution of the script spending the second output of `3a6f`.

Table B.1: Examples of script executions. In each table, the first column indicates what the Script interpreter read from the script, the second one describes the operation performed, and the third one the state of the stack after the interpreter has executed said operation, where the leftmost item corresponds to the bottom of the stack.

that will simply be denoted `data`. Thus, the verification script is `OP_FALSE data OP_HASH160 hash2 OP_EQUAL`; Table B.1b shows its execution.

However, this second output breaks a security recommendation. It is known as a transaction puzzle, where the only thing needed to claim the funds is to find some arbitrary data that is hashed to a given value. It does not include signatures and thus, it can be easily spoofed if it is broadcast before a miner includes it in a block: anyone receiving that transaction can create a conflicting transaction that uses the same input to transfer the funds to another address and broadcast the latter instead. This is a somewhat unusual double-spend setting because the attacker is a third party rather than the buyer, but it looks exactly the same to the network. Incidentally, there is another major security issue with that transaction which is linked to address reuse and is thus out of scope here.

Appendix C

Networking specification

There are (at least) four types of networking related to Bitcoin: the interactions between nodes through the peer-to-peer network, the interactions between nodes through specialized parallel networks (*e.g.* inside mining pools), the interaction between a user and its wallet (*e.g.* to check if a transaction has been confirmed), and, finally, the interactions inside the community (*e.g.* through the community-driven development process). The last two are out of the scope of this document, and the second only has some importance when describing some rational or even malicious behaviours found in the network, in Section 2.3.5.

This appendix builds upon the developer reference and documentation [Ref; Doc] whose claims were checked against the source code of version 0.12.1 of Bitcoin Core [Core], to describe the networking behaviour of the reference client. Many complementary details and figures, such as the byte-level description of all types of messages, can be found in Appendix D. A number of operations performed by Core are not mentioned here, mostly because they are either validity checks (e.g. verifying that a peer runs a protocol version that supports an operation before sending it the corresponding request) or synchronization control between multiple threads.

All Bitcoin messages share some similarities: they are exchanged over TCP and have the same header. The payload, or absence thereof, depends on the message type. More details are given in Appendix D.1. Appendix C.6 describes the multi-threaded infrastructure used by nodes to exchange messages over the network. When, in the following, a node is said to *broadcast* a message, it actually hands it out to this structure, which takes care of sending it.

C.1 Connection management

Alice has three connection modes: default, *connect*, and *addnode*. In the default mode, she constantly tries to establish new connections based on her database of addresses: a specific thread uses two `while(true)` loops to choose a candidate neighbour and try to contact it. In the outer loop, it sleeps for half a second, computes the set of subnets containing a neighbour of Alice, lets the inner loop find

an appropriate address in the database and, finally, tries to establish a connection with it if the inner looper succeeded.

To select an appropriate address, the inner loop randomly picks in Alice's database and checks several conditions: the candidate address must be valid, not belong to the same subnet as any already established neighbour, not be a local address, not belong to a subnet that the user has blacklisted, the corresponding node must be known to provide the minimum required network services[1], must not have been tried for at least 10 minutes unless the loop has made at least 30 unsuccessful iterations and must use the default port (8333 for the main network) unless the loop has made at least 50 unsuccessful iterations[2]. The implementation actually also enforces that the address must be known to provide the network services relevant to Alice unless the loop has made at least 40 unsuccessful iterations but those are exactly the minimum required services and this test is currently redundant. The process of randomly picking an address is detailed in Appendix C.2.

The *connect* mode is actually simpler: when the user specifies a set of addresses to connect to, Alice's connection thread constantly loops over this set to try and establish connections, using a bounded linear back-off mechanism to wait up to 5 seconds between two consecutive connection attempts.

Finally, in the *addnode* mode, Alice runs two connection threads. The first one runs according to its mode of operation as described above and a second one loops over a user-specified set of addresses and behaves almost as the *connect* mode, the most notable difference being in the sleeping periods (0.5 s between each connection attempt and 2 minutes between each iteration over the set of addresses).

When a connection has been established, most messages can be sent by any of the two endpoints. The main exceptions to this rule are the `version` and `getaddress` messages, whose asymmetry are specified in their descriptions, respectively hereafter and in appendix C.2.

As soon as Alice has established an outbound connection to Bob, she sends him her `version` message, whose payload is described in Appendix D.3.1. Upon reception, Bob decodes it to determine if he wants to maintain the connection and how it should be handled. Thus, he drops it if Alice's protocol version is obsolete or if the random nonce is equal to one he has just sent (indicating that he is trying to connect to himself), and the message is rejected if Alice had already sent one. He determines whether Alice wants him to load a Bloom filter to relay transactions (see Appendix C.4), updates the set of addresses on which he can be reached (which can be useful when Bob is behind a NAT and does not know it), sends back both his `version` and `verack` messages, checks whether Alice is a full node or only runs in SPV mode (see Appendix C.4 as well), takes note of the protocol version to use for this connection, and potentially marks Alice's address as good in his address manager (see Appendix C.2) and recomputes the median network time offset, the

[1] That is, it must be able to propagate blocks and transactions.

[2] This is to mitigate a DoS attack that could be performed by advertising the address and port of a server not related to Bitcoin.

median of the offsets between his local clock and a subset of the time stamps it has received in `version` messages. This offset is used in many occasions to check the validity of time stamps.

Then, Alice handles slightly differently Bob's `version` message: she also verifies that their protocol versions are compatible, that Bob only sent it once, whether he wants her to load a Bloom filter and whether he is a full node. If so, Alice marks him as one of her preferred download peers, under the assumption that Bob has less chances of being a malicious node if he was already in the network and she had to contact him to establish a connection than if he contacted her. She also makes sure to use the right protocol version for this connection, sends him both a `verack` and a `getaddress` message and marks his address as good in her address manager and, finally, recompute the median network time offset.

Reception of the `verack` message does not depend on the connection orientation: in both cases, the receiver marks the sender as connected as sends back a `sendheaders` message. When receiving a `sendheaders` message, nodes change the way they advertise blocks to the sender, as described in Appendix C.3.

Every two minutes, Alice sends a `ping` message to all of her neighbours, each containing a random nonce. When Bob receives it, he sends back a `pong` message containing the same nonce. When she receives it, Alice updates her knowledge of the round-trip time (RTT) between Bob and her if the nonce matches the one she sent, which may not be the case (e.g. when Bob's `pong` is the response to an older `ping`). This exchange also serves as a keep-alive for the connection.

The last part of connection management is termination. Bitcoin does not implement any equivalent to TCP's FIN handshake, which means that nodes are never informed that a neighbour has closed the connection. There are a few reasons why Alice may want to disconnect from Bob. The most obvious one is when Alice is shutting down. Blatant malicious behaviour from Bob is another one but, depending on the offence, disconnection may not be instantaneous: an oversized message (more than 4 MB) leads to immediate termination, while sending a second `version` message only gives 1 *misbehaving point* and a neighbour only gets banned upon reaching a total of 100 of them. However, once banned by Alice, Bob needs to wait 24 hours before being able to re-establish connections with her. Different misbehaviours give different misbehaving points and some, particularly those related to filter management, lead to immediate ban.

Finally, Alice may drop her connection with Bob if Carol tries to establish a new one with her and all of her inbound connection slots are taken. In that case, she applies the following logic to choose a neighbour to evict: from her set of neighbours, she withdraws her outbound ones, then sorts the set by the hash[3] of her neighbours' subnet suffixed by a random salt generated during initialization (which makes this eviction process unpredictable, assuming that her random number generator is good enough) and withdraws the last four members from the set, then the eight members with the lowest RTT, then the half of the remaining set with which the connection

[3]Here, only a single SHA-256 hash is performed.

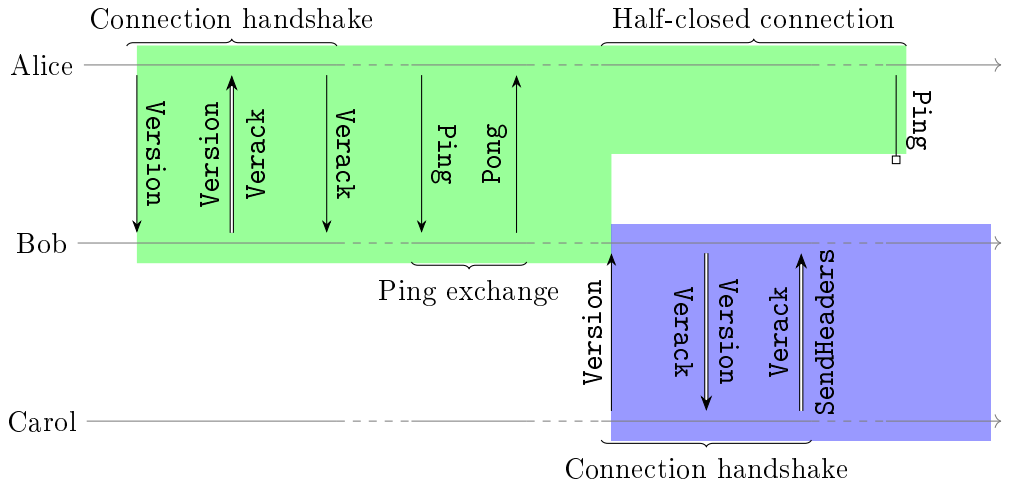

Figure C.1: Example of message flow related to connection establishment and management. Double lines indicate that two messages are sent in a row: as opposed to TCP's piggy-backed ACK, they are sent independently as two distinct replies to a single message.

has been up for the longest time. Finally, she keeps only the subnet that maximizes the number of neighbours still in the set and picks the one with which the connection is the youngest, unless there is only one neighbour left in the set. This procedure, which may seem overcomplicated, is devised to maximise Alice's connection to the network while preventing her from being isolated by an attacker who would try to take up all of her inbound connection slots (and possibly do something nasty to her outbound connections, because otherwise she would not actually be isolated from the network).

Bob will realise that Carol has closed their connection, independently of who initiated it, when one of the following time out scenarios occurs: she sent no message during the first 60 seconds of the connection, he did not succeed in sending her or receiving from her any piece of data, or getting a correct `pong` reply from her in more than 20 minutes. When any of these events happens, he assumes the connection to be dead and closes it.

Figure C.1 summarizes this process. Alice opens a connection with Bob and they perform the Bitcoin 4-way handshake. Later, Alice sends a `ping` to Bob, who replies. When Carol tries to open a connection with him, he decides to let go of Alice, who has no way of knowing it right away. The `ping` time-out is but an example of signal to close the connection.

C.2 Address management

There are two sides to managing addresses in a network: how nodes handle those they know, and how they exchange them between each other. For both of them, Core uses a randomized approach to mitigate, mainly, fingerprinting attacks. Finally, a third operation is required by Core: the random selection of an address when establishing a new connection. Core runs on top of IPv4 and IPv6 depending on their availability and can use Tor [DMS04].

Alice's address manager is a set of 1280 buckets of size 64: 1024 are used to store addresses that are known but have never been tried (called "new" buckets), and the other 256 are used to store addresses that have been tried ("tried" buckets). It is designed to be robust against Sybil attacks.

When Alice receives Bob's address from Carol, she tries to add it in her address manager. This operation fails if the address is not routable (e.g. if it belongs to a private subnet such as 192.168.0.0/16 different from that of Alice). Otherwise, she creates or updates a list of information regarding Bob, including the services he advertises and a time stamp whose value is described in Appendix A.7. If the address was already known but had never been tried and has a more recent time stamp than the one previously reported, appears less than 8 times in the database, and succeeds in a Bernoulli trial with probability 2^n where n is the number of times it already appears in Alice's database, or if it simply is new to Alice, it is added to a new bucket.

To that end, the address manager first computes four hashes on inputs including its own secret key, Bob's and Carol's respective subnets and some modular reductions to obtain a bucket index. Then, it selects a position in the bucket by using two other hash computations based among other on the bucket index and Bob's subnet. Finally, it inserts Bob's address at the selected position in the selected bucket either if it was empty or if the address previously there was not interesting enough (i.e. with a time stamp too old, too far away in the future or with which too many consecutive connections attempts have failed).

The main way for Alice to get addresses is to ask her neighbours to share parts of their databases. This is done through a `getaddress` message, which can only be sent in an outbound connection. As described in Appendix C.1, Alice sends it to all of her outbound neighbours during the `version` handshake.

When Bob receives a `getaddress` message from Carol, he ignores it if Carol is an outbound neighbour. Otherwise, he randomly picks 23 % of the addresses he knows (up to 2500), and sends them back in as many `addr` messages as needed, each one containing up to 1000 entries. Those entries contain, besides the IP address and port number, information about the services Bob thinks they provide and a time stamp.

Besides this query/response behaviour, nodes can also, under some conditions, push addresses without an explicit query. First, for each of her neighbours, Alice keeps a (future) time stamp after which she advertises her own address. When she does, she computes the next duration to wait before doing it again as a random

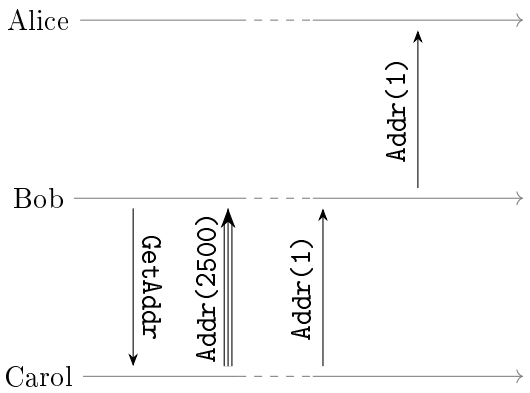

Figure C.2: Example of message flow related to address management. In response to Bob's `GetAddr`, Carol sends three `Addr` messages containing respectively 1000, 1000 and 500 addresses.

variable following a Poisson process with mean 576 minutes (9 h and 36 min). She also advertises it right before sending the `getaddr` message during connection establishment with an outbound neighbour. When receiving an `addr` message from Carol, Alice may forward its content to up to two neighbours. This happens if the message contains at most 10 entries, and Alice has not requested a `getaddr` message from Carol or Carol has previously sent an `addr` that was not full (i.e. less than 1000 entries), and only the entries with a time stamp at most 10-minute old, that are routable (i.e. not belonging to a private subnet such as 192.168.0.0/16) are relayed. The neighbours to which Alice relays those addresses are picked based on hashing computations, a random salt generated at initialization time and a time stamp with a 24-hour granularity: this way, Alice can get a reasonable idea of which addresses those neighbours know to avoid sending them again and again the same ones.

Figure C.2 summarizes this process: Bob is connected to Alice and opens a connection to Carol (not shown) and asks for a list of addresses. She replies with the maximum number of addresses (assuming that she has more than 10 870 addresses stored in her address manager). After a while, she advertises her own address to Bob, who forwards it to Alice.

C.3 Block and transaction propagation

All of the above should not let one forget Bitcoin's purpose: replicate a ledger over all nodes involved in the network. Thus, the only messages that are actually at the core of Bitcoin are the `block` and `tx` ones, while the rest can be seen as support functions to get them where they need to be. Transactions, blocks, their respective validity checking procedures and the specific initial block download procedure are described in Appendices A.3 and A.4; this section focuses on the messages exchanged

by approximately synchronized nodes when a new block or transaction is propagated through the network.

The main way for Bob to learn that Alice has blocks and/or transactions to send him is by receiving `inv` messages from her, which can advertise up to 50 000 hashes each. When Bob receives such a message, he iterates over its entries and deals with them based on their type if he does not already have the corresponding data (in which case he simply drops the inventory):

Block Unless it has already been requested from another neighbour, Bob asks Alice both for all the headers between his current tip and the advertised block (in case a few are missing between them through a `getheaders` message) and the block itself, unless Alice is already transferring 16 blocks;

Transaction Unless it has already asked one of his neighbours for the corresponding transaction, he asks her for it.

In both cases, Bob's query is made through `getdata` messages, containing at most 1000 inventory requests. When Alice receives one, she replies with the appropriate amount of `block` or `tx` messages to supply the data, handling pathological cases (e.g. `getdata` requests for data she doesn't have) either by ignoring them, sending `notfound` messages or terminating the connection.

When Bob receives a valid transaction, it relays it by sending the corresponding `inv` to his neighbours. He then iterates over his set of orphan transactions to recursively validate those that were waiting for this input (and relay the newly valid ones as well). If the transaction is invalid because it misses inputs, it is kept as an orphan but not relayed while waiting for the parent transaction to be received. At most 100 orphan transactions are kept at the same time: when the set grows bigger, random elements are picked and pruned.

When Bob receives a block, the validation process is quite more complex; the result is that he broadcasts it to his neighbours if it becomes the tip of his local blockchain, along with all the blocks on which it is built that were not part of the main chain before (which only exist in case of fork).

In both cases, broadcast is done using the same three-way exchange: `inv`, `getdata` and actual data message[4]. However, there is one more trick to `inv` broadcast: when preparing a batch of `inv`'s for Alice, Bob include the latest blocks if he doesn't know whether Alice already has them and determines whether or not the message should include all transactions (this happens every few seconds, the delay is generated through a random Poisson process with a mean of 5 seconds). For every incomplete batch, each transaction has a 25 % chance of being included: the hash of the xoring of its hash with a random salt (generated at node initialization) must have two trailing zeroes (this means that if a transaction is not selected for a batch of incomplete `inv`'s, it will never be selected before the next batch of complete ones).

[4]The term "three-way" exchange is used for blocks even though it actually comprises 5 messages because they are sent in three batches.

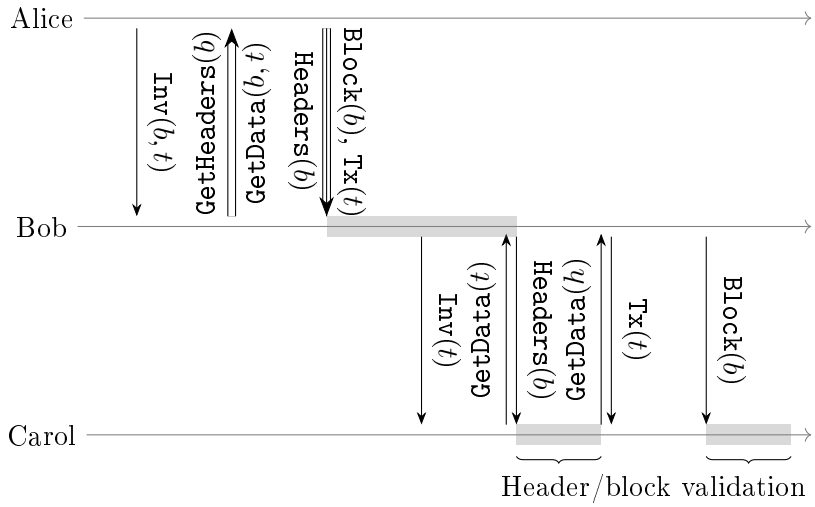

Figure C.3: Example of message flow related to data propagation. Alice and Carol are two neighbours of Bob that are not neighbours of each other. Alice advertises a block b and a transaction t.

From v0.12.0 on, Bob can also advertise blocks to Alice by directly sending her the headers, instead of an `inv`, assuming she has declared being interested in that option. In that case, right before preparing a batch of `inv`'s, Bob looks for the first block he has that Alice doesn't, and sends a `headers` message containing its header and all the following ones. Upon reception of a `headers`, Alice validates each entry and stores it in her chain of headers (a sub-part of the blockchain) so that she can skip this part of the block validation process when receiving the rest of the block and only requests the full blocks that actually make the main branch of her blockchain grow (which includes switching main branch had she opted for the loosing branch of a fork) that she has not already requested.

This describes how blocks and transactions are propagated over the network once someone has started transmitting them. It usually is up to the signer of a transaction to make sure it is propagated: to that end, the newly signed transaction is added to the list of those that need to be relayed. Every so often (i.e. after a random amount of time of up to 30 minutes), Bob rebroadcasts those that are older by more than 5 minutes than the last block and still unconfirmed. As regards blocks, whenever Bob's local blockchain tip is changed, every block newly included in the active branch (which may be more than one only in case of fork) is advertised to all neighbours using the regular block propagation mechanism. This includes newly found blocks.

Figure C.3 summarizes this process: Bob is connected to Alice and Carol, when Alice starts broadcasting a block and a transaction. He asks for both and broadcasts each as soon as he has validated them. Compared to the time needed to verify a usual block, verifying a single transaction seems instantaneous. Moreover, he agreed

with Carol during the connection handshake to advertise blocks by sending directly their headers to let her validate it before requesting the rest of the block.

C.4 SPV nodes

Peers with limited bandwidth, processing power or energy supply can also run in Simplified Payment Verification (SPV) mode. Basically, when doing so, the peer trusts the network to validate blocks and transactions and only performs minimal verification of what it receives: it may assume the role of wallet but only partially handle those of router and blockchain store.

This mode of operation is advertised in the `version` message; nodes will drop outbound connections to SPV nodes but will accept inbound ones. Right after connection establishment, the SPV client, Bob, will send his neighbour, Alice, a Bloom filter[5] [Blo70]. Then, whenever Alice can relay a transaction to Bob, she will first check it against Bob's filter to see if it has a chance of interesting him and, if not, will not send it to him.

Block transmission is also modified: instead of regular `block` messages, Alice will send `merkleblock` ones. They contain the regular header and the list of hashes needed to verify that the transactions Bob wants to know about are included in it as advertised; since they do not contain the actual transactions, Alice also sends as many `tx` messages as needed along with each of them. How this verification works is described in Appendix A.4.3.

The reference client does not include an SPV mode: it cannot handle receiving `merkleblock` messages, and would reject any coinbase transaction sent as a loose transaction, which would happen if it was interested by one.

C.5 Additional sources of complexity

Most of Appendix C has described the regular operations of Core when left alone in charge of every decision to make. However, there are many customization options available to the user which make the decision-making process quite more complex. Examples include the *addnode* and *connect* options described in Appendix C.1 but also white and black listing of addresses and networks, upper bounding the amount of data uploaded to the network and changing most of the internal constants such as the duration of banishment. Moreover, Bitcoin clients usually have to accept interactions with the user, at least to generate new transactions on demand: this is the only thing that cannot be left entirely up to the computer.

All in all, Core is a complex program: running

```
cloc $(git ls-files)
```

[5] A Bloom filter is (here) a data structure allowing to check efficiently if a transaction is definitely not or probably of interest. More details are provided in Appendix A.8.

on v0.12.1 of its repository cloned from Github [Core] gives a total of 312 426 lines of code, out of which 97 537 are in C++ files and header files (and 167 701 lines are Qt Linguist ones, coming from the several available translations of the graphical interface), plus barely 16 276 lines of comments. Thus, it would be an extremely difficult and tedious task to provide a complete analysis which would probably be outdated before being complete.

There are two other important reasons explaining the complexity of the Bitcoin network. First, Core is far from being alone in the network, as reported in Appendix D: it barely represents 46.98 % of the nodes. Though its two main competitors are older versions of itself, it still makes for complex backward-compatibility handling. Given its Open-Source status, it is easy for anyone to modify the code and alter the way some operations are performed, as was done to perform an experiment during this work. This goes towards the trustless model: peers have no way of knowing what their neighbours do, and can only assume the worst. Then, the Bitcoin network is not even the only way for data to be propagated: pools tend to advertise the blocks they find on their websites, making it easy to fetch the information directly through HTTP requests. Similarly, trackers such as Blockchain.info [BC.I] have developed APIs to query them for blocks and transactions. There are also specialized parallel networks [Cor16], focusing on high-speed propagation, which create invisible edges with very unusual characteristics in the graph. Though the principle is good, as decreasing propagation delays increases the effective computing power by helping miners receive the latest block quickly, they may have unwanted side effects. Indeed, BlueMatt's Bitcoin Relay Network [Cor16] was centralised, which tends to go against Bitcoin's model; moreover, the impact on the global Internet congestion may be non-negligible if largely adopted, especially those, such as BlueMatt's FIBRE, that are based on UDP, a protocol known for its lack of congestion control. However, analysing whether this would actually have any impact on the Internet infrastructures is completely out of the scope of this work.

C.6 Interfacing application and transport layer

Core [Core] uses a multi-threaded approach to network communications. It basically splits the application layer in the TCP/IP model in two: the upper layer handles Bitcoin messages and the lower layer handles serialized data that the TCP layer can handle as is. Each layer is governed by a thread that continuously loops and switches between sending and receiving data.

The lower layer is managed by the Net thread. It relies on two buffers per neighbour, both containing serialized data: one is for data to send, the other for received data. During each loop iteration, it closes the connection to nodes flagged by the rest of the program and deletes them from memory when there is no pointer to them left in the program, and then service each socket. First, it handles the listening ones (or, usually, *one*): for each socket listening for connection attempts, it tries to accept new attempts and add the other end-point as a neighbour (the

upper layer is in charge of handling the Bitcoin connection handshake described in Appendix C.1). Then, each socket corresponding to an established connection is assigned exactly one of three statuses: sending (if the associated send buffer is not empty), receiving (if there is no buffered message ready to be handed to the upper layer), or idle. These states are mutually exclusive, and assigned in that order of precedence. Error management apart, sending (respectively receiving) sockets that are ready to send (respectively receive) data do just that, pushing to (pulling from) the network serialized data from (to) the appropriate buffer. After that, inactivity checking is performed as described in Appendix C.1.

The upper layer is managed by the MsgHand thread. It relies on several buffers per neighbours, each associated to a specific type of message to send (addresses, inventories,...). During each loop iteration, it iterates over all neighbours to process received messages, decides to skip the final sleeping phase if it still has data to receive from any neighbour and can hand over more data to the lower layer destined to that same neighbour and, finally, transfer all buffered outbound messages (which includes generating new ones) to the lower layer. Assuming that it did not decide to skip it, it then sleeps for 0.1 s.

Depending on the message being processed, the function deciding to send a message can either just buffer it and let the MsgHand thread take care of pushing it to the lower layer or do it itself. Generally, Bitcoin follows an optimistic approach: when a message is pushed to the serialized sending buffer, it immediately tries to send it to the corresponding neighbour and only lets the Net thread take care of what failed to be sent that way.

Figure C.4 summarizes most of this organisation. It is, however, far from displaying everything. Specifically, the sleeping periods, validity checks, and interactions with the local blockchain and the user are not shown.

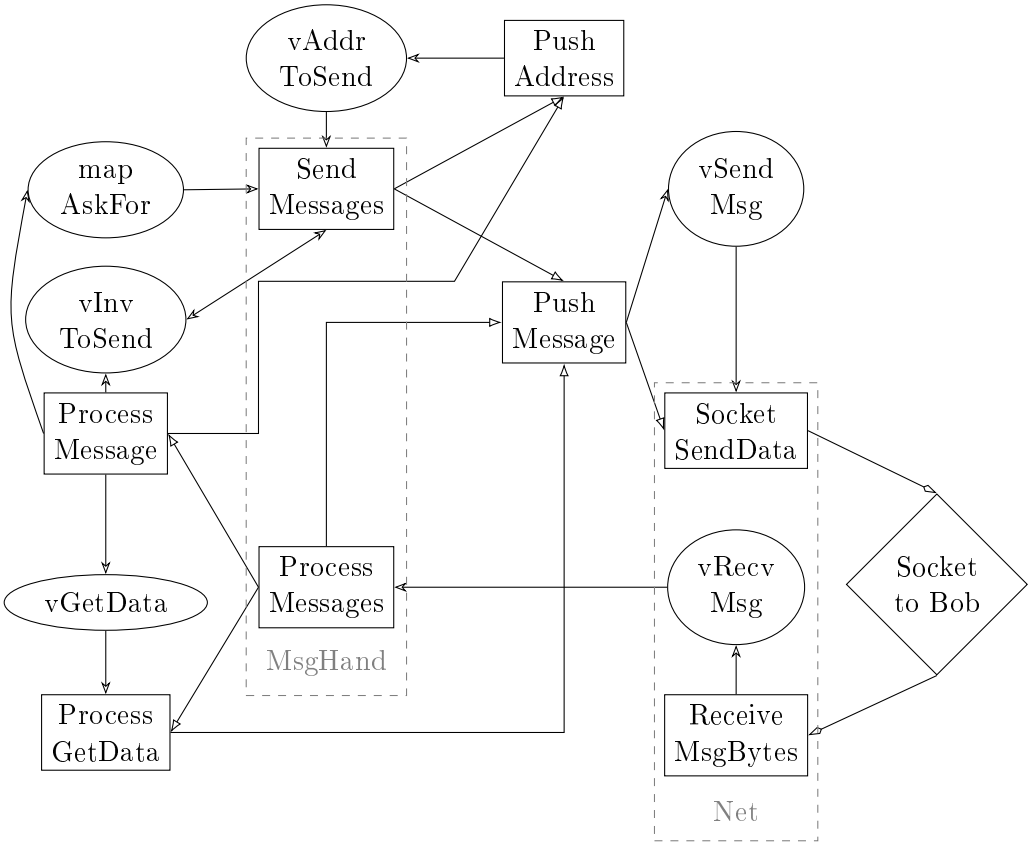

Figure C.4: Structural organisation of the interface between a node and the socket representing one of its neighbours. Functions are represented as rectangles, buffers as rectangles and the TCP socket connecting to Bob as a diamond. Arrows between functions represent calls, the others data flows. The functions called by each thread are shown with dashed rectangles. The mechanisms for `alert` messages are not shown and some edges actually represent a chain of calls.

Appendix D

Bitcoin messages

While Appendix C described how nodes use the Bitcoin messages to exchange and propagate information in the network, this appendix focuses on the exact content of the messages defined by Bitcoin's protocol in its version 70 012. Again, the three main references used for this section are the Bitcoin developer reference [Ref], describing without any guarantee of accuracy a reference client slightly older than that used for this work (Satoshi 0.12.1 instead of 0.13.0rc1), the unofficial Bitcoin developer documentation [Doc] and the source code of versions 0.12.1 of Bitcoin core [Core]: though version 0.13.0rc1 was used for this work, Bitnodes [BN] reported on August 7$^{\text{th}}$, 2016 at 14:44:36 GMT that 46.98 % of nodes were running v0.12.1 of the reference client while barely 117 nodes (i.e. 2.2 % of the 5344 referenced nodes) were running v0.13.*, out of which some advertised the non-official v0.13.99. Other somewhat popular clients were v0.11.2 (8.94 %) and v0.12.0 (8.68 %) of the reference client, and any other client (including non- reference ones) was run by less than 5 % of the network as seen by Bitnodes.

It is organized as follow: first is described the shared header of all the messages, then the types used to exchange data between nodes and finally the control messages used, among others, to negotiate the parameters of each connection. All labels in this section indicate byte count, rather than bits. The `MemPool`, `FilterClear`, `GetAddr`, `SendHeaders` and `VerAck` messages are not described here as they do not include a payload.

D.1 Header

All Bitcoin messages share a common format: a 24 B header and an optional payload, as shows Figure D.1. The header contains 4 fields, as follows:

1. The magic string defines the network to which the message belongs; as of August 2016, five have been officially standardized, out of which four are dedicated to experimentations. The main network uses `0xf9beb4d9`.

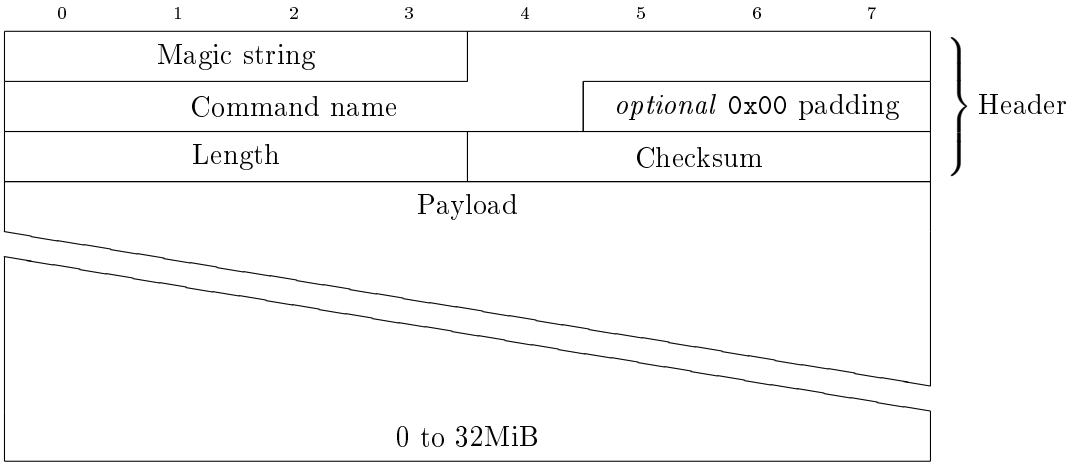

Figure D.1: General format of a Bitcoin message.

2. The command name contains the type of message as a string. It is padded up to 12 bytes with null-characters (`0x00`).

3. The length is that of the payload, in bytes; the maximum allowed is, as of August 2016, 32 MiB.

4. The first four bytes of the double SHA-256 hash of the payload, or of the empty string if there is none, are used as a checksum.

D.2 Data messages

D.2.1 Inv, GetData and NotFound

The `inv` message contains a payload of $36k + \text{cmpct}(k)$ bytes, where k is the number of inventory objects advertised by the sender (between 1 and 50 000) and is the first field of the message. The rest is a list of inventory objects, whose structure is shown in figure D.2. The fields are as follows:

1. Type indicates whether the inventory is a transaction (1), a block (2) or a filtered block (3, see Appendixes D.2.3 and D.3.3), the latter being forbidden in `inv` messages.

2. The hash is that of the object being advertised, e.g. the double SHA-256 hash of the header of the advertised block.

An `inv` message can contain up to 50 000 inventory objects: if more inventories need to be transmitted, several `inv` messages can be sent at a time.

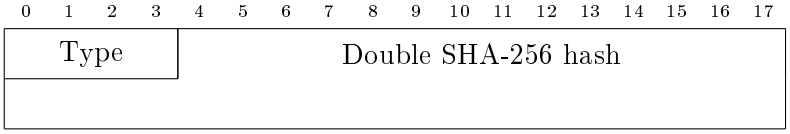

Figure D.2: General format of an inventory.

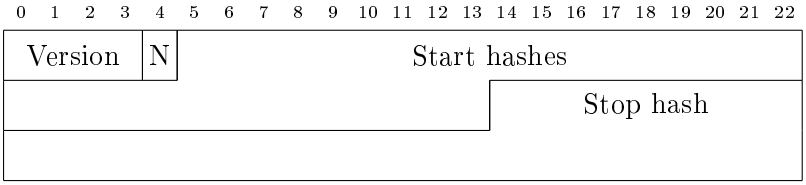

Figure D.3: Format of a `getblocks` message providing only one header hash.

The `getdata` and `notfound` messages follow the same format, respectively requesting or declaring not being able to send the contained inventory objects: only the command name of the message header is different.

D.2.2 GetBlocks, GetHeaders

The `getblocks` message contains a payload of $36 + 32k + \text{cmpct}(k)$ bytes, where k is the number of hashes provided by the querier (at least one, the only limit being the maximum length of a message). Figure D.3 shows a `getblocks` message with only one hash provided. Its fields are as follows:

1. Version repeats the protocol version number, already sent in the `version` message.
2. N is the number of start hashes provided by the querier.
3. Start hashes contains as many 32 B block hash as indicated by the previous field, sorted by decreasing block height.
4. Stop hash contains the last block hash that the querier requests, or all zeroes for no limit, in which case the receiver will send back at most 500 block hashes.

D.2.3 Block, Headers and MerkleBlock

The `block` message contains a single block b as its payload, with a size of $80 + \text{cmpct}(|c(b)|) + f(c(b))$ bytes, where $f(c(b))$ is the total size of the transactions it contains. As mentioned in Appendix A.4.4, the payload cannot be bigger than 1 MB. See Figure D.4 for the format of a block header and (the description of the fields is given in Appendix A.4.1).

The `headers` message contains a payload of $81k + \text{cmpct}(k)$ bytes, where k is the number of block headers in the message (upper-bounded by 2000). Its fields are as follows:

1. A compact size unsigned integer count of entries;
2. Each of those entries are made of:
   a) A 80 B block header;
   b) A `0x00` byte, signalling that the header does not contain any transaction.

The `merkleblock` message contains a payload of $84 + 32t + \text{cmpct}(h) + \lceil \frac{h}{8} \rceil + \text{cmpct}(\lceil \frac{h}{8} \rceil)$, where t is the number of hashes needed in order to verify that the transactions of interest are at their advertised place in the block's Merkle tree. Figure D.4 shows a `merkleblock` message with only one hash (which only happens for empty blocks). Its fields are as follows:

1. A 80 B block header;
2. A 4 B field counting the total number of transactions in the block;
3. A compact size unsigned integer count of the number of hashes provided to verify the Merkle tree;
4. As many 32 B hashes as announced, corresponding either to transactions or Merkle nodes;
5. A compact size unsigned integer count of the number of flag bytes;
6. As many flag bytes as announced, used to verify the Merkle tree as described in Appendix A.4.3;

The transactions of interest are not actually sent through the `merkleblock` message but as separate `tx` messages, described in Appendix D.2.4.

D.2.4 Tx

The `tx` message contains a single transaction. See Appendix A.3 for a detailed description.

D.3 Control messages

D.3.1 Version

The `version` message contains a payload of $85 + k + \text{cmpct}(k)$ bytes, where k is the length of the sender's user agent. Figure D.5 shows a `version` message with a variable-length part, the L and User agent fields, of 15 B. Its fields are as follows:

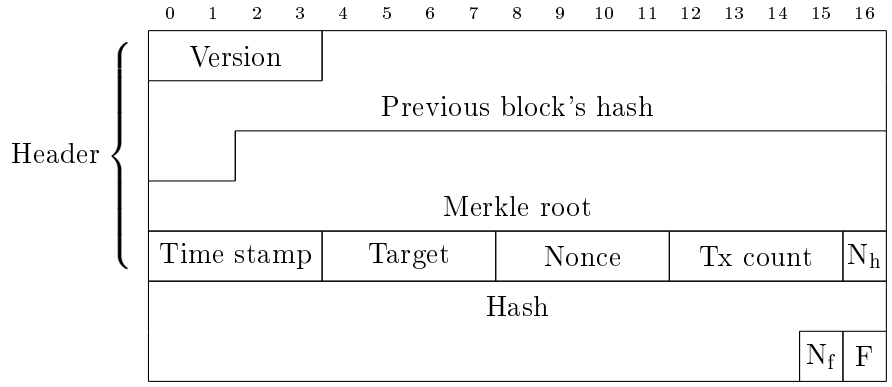

Figure D.4: Format of a `merkleblock` message providing only one hash. The first five rows describe the header of a block, identical for Merkle and regular ones; TxCount and N_h are not part of it.

1. A 4 B Version field indicating the highest protocol version supported by the sender. For v0.12.1, the latest version is 70 012.

2. An 8 B bitmask indicated the services provided by the sender. The 0^{th} bit is used to show that the node can send blocks (as opposed to SPV nodes), the first indicates a service that is not implemented in the reference client, the second that the node accepts to load and abide by Bloom filters (see Appendix D.3.3). Finally, bits 24 to 31 are reserved for experiments and everything else is reserved for future use[1].

3. An 8 B time stamp loosely used for clock synchronization in the computation of the median network time offset (see Appendix C.1);

4. Then, the header include the same three fields describing first the receiver, then the sender. When prefixed with "Rcv", it describes the receiver as the sender perceives it; when prefixed with "Sdr", it describes the sender:

   a) An 8 B service which has the same meaning as the previous Services field (Sdr Services actually being redundant);

   b) A 16 B IPv6 address; IPv4 addresses are mapped as per RFC 4291 [RFC4291];

   c) A 2 B TCP port number;

5. An 8 B nonce, used to detect connections to self;

6. A compact size unsigned integer count L of bytes in the following field;

7. A User agent that gives information about the Bitcoin client used by the sender, such as `/Satoshi:0.12.1/` for v0.12.1 of the reference client;

[1] Bit 3 is already reserved for Segregated Witness, an option that will be included in a future version of the reference client.

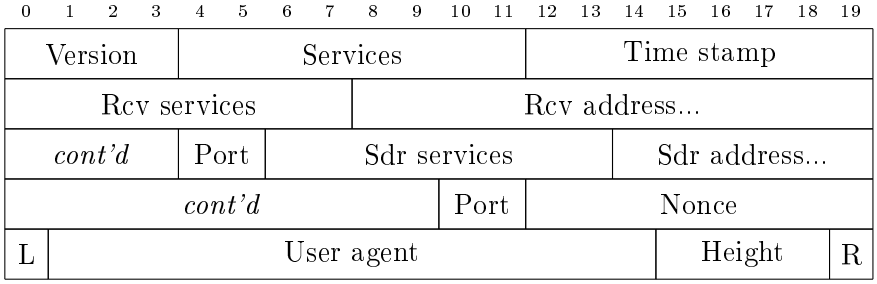

Figure D.5: Payload of a `version` message with a 14 B user agent. "Rcv" fields give information the sender knows about the receiver, the "Sdr" ones are about the sender.

8. A 4 B height, the length of the longest branch of the sender's local blockchain;
9. A 1 B *R*elay flag: if set to 0, the sender wants to send a Bloom filter to the receiver before being sent `inv` and `tx` messages; see Appendix D.3.3 for more details.

D.3.2 Addr

The `addr` message contains a payload of $20k+\text{cmpct}(k)$ bytes, where k is the number of addresses in the message, up to 1000. Just as the `headers` message, it is a list of network addresses prefixed by its number of entries, which are all formatted as follows:

1. A 4 B time stamp, described in Appendix A.7;
2. An 8 B service bitmask as described in the `version` message;
3. A 16 B IPv6 address or IPv4-mapped IPv6 address;
4. A 2 B port number.

D.3.3 Bloom filters

There are three message type related to Bloom filter management: `FilterLoad`, `FilterAdd`, and `FilterClear`. The latter does not have a payload. The `filterload` message contains a payload of $9 + k + \text{cmpct}(k)$ bytes, where k is the byte length of the data used to initialize the filter. Figure D.6 shows a `filterLoad` message with 8 B of data to be used. Its fields are as follows:

1. A compact size unsigned integer count of bytes in the following field;
2. Filter data stored in a byte vector;

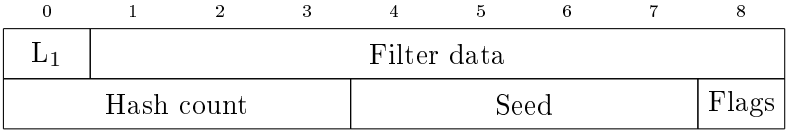

Figure D.6: Payload of a `filterload` message with 8 B of filter data.

3. A 4 B number of hash functions used by the filter; the maximum accepted value is 50;
4. A 4 B salt for the seed used in the hash function of the field;
5. A 1 B flag describing how and when to update the filter: if the least significant bit is set, the filter is updated whenever a transaction that matches is found; if only the following bit is set, the filter is updated only if the script of a matching transaction pays to a public key (or set thereof, in case of multisignature scripts). The other bits are reserved for future use.

The `filteradd` message contains a payload of $k + \text{cmpct}(k)$ bytes, where k is the byte length of the data to add to the filter, which is upper bounded by 520 B. The payload only contains this element as a byte vector, prefixed by its length as a compact size unsigned integer.

D.3.4 Ping and pong

The `ping` and `pong` messages contain an 8 B payload, a single nonce.

D.3.5 Reject

The `reject` message contains a variable-size payload. Figure D.7 shows a `reject` message with variable-length fields set to valid arbitrary sizes. Its fields are as follows:

1. A compact size unsigned integer count of bytes in the following field;
2. The command name of the rejected message, without null padding (2 B to 12 B);
3. A 1 B error code;
4. Another compact size unsigned integer count of bytes in the following field;
5. An ASCII explanation of the error, for debugging purposes (up to 111 B);
6. Extra data, depending on the type of the rejected message and the error code; usually, either empty or set to the hash of the rejected object (0 or 32 B);

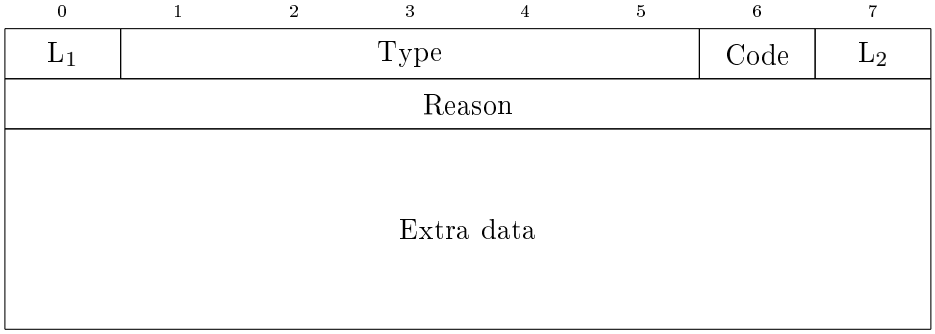

Figure D.7: Payload of a `reject` message answering to a `block` message with an 8 B reason and 32 B of extra data.

D.3.6 Alert

The `alert` message contains an encapsulated payload. The outer part of the payload is as follows:

1. A compact size unsigned integer count of bytes in the following field;
2. The inner payload, see below;
3. A compact size unsigned integer count of bytes in the following field;
4. The DER-encoded signature of the alert, produced by a developer with a special alert key.

Figure D.8 shows the inner part of an `alert` message with variable-length fields set to valid arbitrary sizes. Its fields are as follows:

1. A 4 B alert format version, still 1 in protocol version 70 012;
2. An 8 B POSIX time stamp indicating at which point nodes should stop relaying the alert;
3. An 8 B POSIX time stamp indicating the expiration time of the alert;
4. A 4 B alert ID;
5. A 4 B threshold: all alerts with ID below it should be cancelled;
6. A compact size unsigned integer count of entries in the following field;
7. A vector of 4 B ID's indicating specific alerts that are cancelled by this one;
8. A 4 B minimum protocol version; this alert does not apply to nodes running a version strictly less than it but they should still relay it;

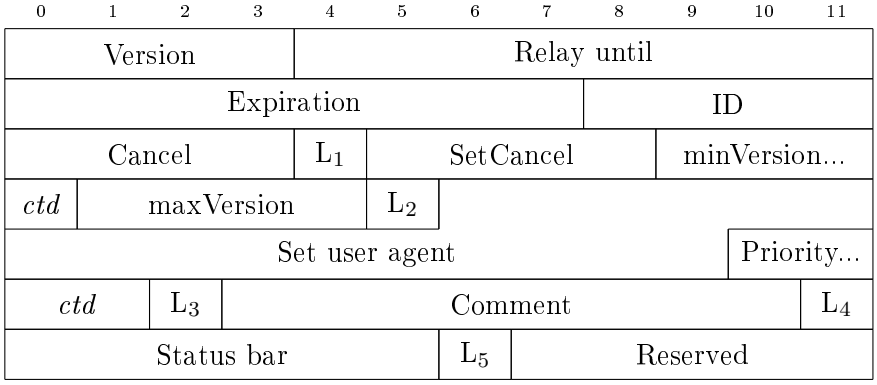

Figure D.8: Payload of an `alert` with variable fields set to arbitrary but valid lengths.

9. A 4 B maximum protocol version; this alert does not apply to nodes running a version strictly greater than it but they should still relay it;
10. A compact size unsigned integer count of entries in the following field;
11. A vector of user agents represented as a variable-length string prefixed by a compact size unsigned integer count of bytes; this alert only applies to nodes running one of those user agents. If empty, it has no effect on the alert (all nodes with an affected protocol version are affected);
12. A 4 B priority over the other alerts;
13. A compact size unsigned integer count of bytes in the following field;
14. A comment string that should not be displayed to the user;
15. A compact size unsigned integer count of bytes in the following field;
16. A comment string that should be displayed to the user;
17. A compact size unsigned integer count of bytes in the following field;
18. A string field reserved for future use.

Note that the `alert` message has been removed as of version 0.13.0.

Appendix E

Glossary

account

Conceptually, an ECDSA keypair used to send money by signing transactions and receive it through the associated address. In our formal model, is used to denote both a transaction output and the input spending it. It is important to notice that these two are differently typed Bitcoin objects and that Bitcoin does not actually defines accounts: the concept was ported from the banking world to simplify analogies.

address

Depending on the context, may either correspond to Bitcoin or network addresses. The former is a bitstring computed from the public key of an account through operations involving hashes among others used to hide the public key for as long as it has not been used. The latter is the network address of a node, as in the combination of IP address and port number that identifies it; it is however supposedly very hard to find which Bitcoin addresses correspond to which network address. See, respectively, Appendices A.6 and A.7.

address manager

Object handling a node's address database as described in Appendix C.2.

altcoin

Describe every (decentralized) crypto-currency that is not Bitcoin, such as Ethereum [But14].

bitcoin

Unit of the currency defined by Bitcoin. Abbreviated BTC, their exchange rate for traditional currencies has had considerable variations since 2008.

Bitcoin Improvement Proposal (BIP)

Document describing a way to modify Bitcoin that is submitted to the community for approval. May or may not be accepted and implemented.

block

Structure containing a list of transactions, a PoW, and a pointer to its parent in the blockchain, found at a rate of approximately one every ten minutes by the miners of the Bitcoin network.

block conflict detection service (BCDS)

Specialised CDS for blocks..

blockchain

Bitcoin's ledger, tree of all the blocks that have been found. The longest branch of the tree (in terms of difficulty) is the consensus one, the blockchain *per se*. Regular transactions are performed on the main chain but there are also two defined specifically for performing tests without perturbing the main one.

coin

See bitcoin.

coinbase transaction

First transaction of a block, it attributes the block reward (minting and fees) to the miner..

compact size unsigned integer

Integer stored on a variable number of bytes. See Appendix A.1 for the detailed description.

conflict detection service (CDS)

Service in charge of detecting and solving conflicts such as double-spending attempts and forks..

denial of service (DoS)

Here, attack consisting in disrupting the access of a node to the network, *e.g.* by flooding it with packets or by not forwarding messages to or from it.

difficulty

Ratio between the maximum target and the current one: if equal to D, a 256-bit-long bitstring chosen uniformly at random has probability $1/(2^{32}D)$ to be below the corresponding target.

distributed hash table (DHT)

Class of distributed system proving a lookup service based on keys and maintained by the network.

double-spend

Attack consisting in emitting two conflicting transactions to get the recipient of one of them to believe that he received funds and get it invalidated by the network when the second one is the one included in a block.

Elliptic Curve Digital Signature Algorithm (ECDSA)

Private-key cryptographic signing primitive used by Bitcoin, defined in [JMV01].

fee

Also called transaction fee. Difference between the sum of inputs of a transaction and the sum of its output, that the miner including this transaction in a block is invited to get control of through the coinbase transaction. Acts as an incentive for miners to try and include the transaction in blocks.

find

A block is said to have been found when a miner finds a nonce such that it solves a PoW. The miner finding a block is rewarded.

fork

Situation with two chains competing to be accepted by the network as *the* blockchain. Happens when two valid blocks are mined on top of the same parent.

hard fork

Fork caused by the propagation of blocks valid for different and incompatible versions of the protocol. Happens *e.g.* when the size limit of blocks is increased: the nodes still using the old protocol will refuse the bigger blocks.

hash

Output of a hash function that takes an input of arbitrary length and outputs a seemingly random bitstring of fixed length. Bitcoin tends to use double hashes rather than simple: 256-bit hashing uses $(\text{SHA-256})^2$ and 160-bit hashing is the result of RIPEMD-160 applied to the SHA-256 hash of the input.

inbound

Related to a connection initiated by someone else. Its other endpoint sees it as outbound.

mempool

Pool of pending transactions maintained internally by each blockchain-storing peer. Stands for *memory pool* as it is mostly kept in main memory.

mine

Action of trying to solve PoWs in order to find blocks.

miner

Agent trying to solve PoWs to generate blocks.

node

Peer of the Bitcoin network maintaining a local blockchain and a mempool, connected to the network and acting as a miner.

outbound

Related to a connection initiated by the node. The other endpoint of said connection sees it as inbound.

peer

Set of software acting as a unique agent. May be any combination of a Bitcoin router, a blockchain store, a wallet and a miner.

pool

Group of miners working together to find blocks. Usually, when one is found, the reward is shared between the pool members and infrastructure according to some predefined rule.

probability density function (PDF)

Function describing the probability that a random variable falls within ranges of values.

proof of stake (PoS)

Process using a blockchain to seed a random number generator in a publicly verifiable way in order to elect the next block finder with the same probability distribution as the distribution of coins in the system.

proof of work (PoW)

Cryptographic puzzle randomly solved. In Bitcoin, consists in finding a nonce such that the double SHA-256 of the block header that includes it has a number of leading zeroes computed by the system in order to set the average generation rate of blocks to ten minutes.

public key infrastructure (PKI)

Trusted infrastructure distributing certificates to entities so that they can prove their identity to other entities.

RACE Integrity Primitives Evaluation Message Digest (RIPEMD)

Family of hash functions. Bitcoin uses the 160-bit version, introduced in [DBP96].

round-trip time (RTT)

Duration between the instant when a device A sends a message to a device B and that when it receives B's response, computed preferably in situations where B does not need to perform heavy operations in order to prepare the response.

router

Role of a peer connecting to the Bitcoin network to receive and propagate data.

satoshi

Smallest division of bitcoins, equal to 10^{-8} BTC.

scriptPubKey

Script included in a transaction output to make sure only the recipient can redeem it. See Appendix B.

scriptSig

Script included in a transaction input to prove that the emitter has the right to redeem it. See Appendix B.

Secure Hash Algorithm (SHA)

Family of (families of) hash functions. Bitcoin uses the 256-bit version, standardized in [FIPS180-4].

Simplified Payment Verification (SPV)

Mode of operation of a Bitcoin node that does not verify the validity of the blocks it receives. See also SPV mine.

SPV mine

Action of mining on top of blocks that have not been validated. Considered an attack on the network.

store

Role of a peer maintaining a local copy of the blockchain.

target

256-bit-long number such that the hash of a block header must be less than or equal to it for the block to be considered valid. See also difficulty.

transaction

Basic structure of Bitcoin transferring the control of some coins from a list of accounts to another one.

transaction conflict detection service (TCDS)

Specialised CDS for transactions..

unspent transaction output (UTxO)

Output of a transaction that has not yet been used as an input by another transaction. Does not qualify outputs with a provably unredeemable scriptPubKey.

wallet

Software handling transactions: tracks those sending funds to the set of keys it manages and signs those requested by the user.