A blockchain-based framework for the management of IP prefix allocations and BGP updates in the Internet

Computer Science Department,
University of Crete

Undergraduate Student: **Ilias Sfyrakis** [csd3078@csd.uoc.gr]
Primary advisor: **Dr. Vasileios Kotronis** [vkotronis@ics.forth.gr]
Supervising professor: **Prof. Xenofontas Dimitropoulos** [fontas@ics.forth.gr]

Abstract

In this thesis, we implemented a prototype blockchain-based framework for the management of Internet resources such as IP prefix allocations and BGP updates, to enable their online validation based on tamper-proof distributed ledgers without any centralized authorities intervening in this management. Current solutions to the security issues that BGP and the Internet generally suffer from, have not been deployed yet mainly due to associated high technical and financial costs. The main goal of this thesis is twofold: (i) to investigate if the state-of-the-art "Internet blockchain" concept proposal could be implemented using practical mechanisms, and (ii) to evaluate its scalability and efficiency using real BGP updates and realistic assumptions. Our back-of-the-envelope calculations show that our blockchain needs O(10) Gbytes for storage, which increases by approximately O(100) kbytes per day, mainly due to the BGP updates that propagate in the Internet. Depending on the complexity of the consensus algorithm that is used for block mining and validation (e.g., Proof of Work) and the aggregation of transactions in blocks, it seems that the blockchain is able to cope with the online management of such resources. However, we still need to evaluate our results further under much higher control-plane message loads and with tens of thousands of participating nodes in the network. We note that despite the fact that we used official BGP reports and datasets for our calculations, even if the estimated results are off by one order of magnitude, manageability and scalability (including storage capacity and speed) can be still maintained.

For the development purposes of our framework, we used Python since it allows for fast prototyping, and it is an easy language to learn. We demonstrate our prototype by setting up a small network of processes that represent ASes which generate transactions in a distributed -overlay- peer-to-peer network. We show the chain, how is it formed, the transactions that are mined into blocks (IP Allocation Assign/Update/Revoke, BGP Update Announcement/Withdrawal), as well as the two main types of state (IP prefix assignment and the AS-level graph per prefix) we hold for the validation of the transactions and how they change due to the addition of valid transactions. We believe that this thesis can help in moving forward the research on the management of IP resources using blockchain-like mechanisms; moreover, it provided me the opportunity to learn about BGP (and its (in-)security), Bitcoin and the Blockchain, as well as program from scratch a useful prototype based on principles of modularity and extensibility.

1. Introduction

1.1. Motivation

BGP (in-)security has been one of the biggest issues of inter-domain routing for decades; e.g., the lack of proper authentication mechanisms has repeatedly resulted in BGP hijacks and other routing attacks [9, 14]. However, changing the protocol (even by considering security “add-ons”) is a slow process, spanning multiple years and tens of thousands of networks; for example, RPKI and route origin validation have only recently started to take off, while the mediation of centralized authorities for the management of RPKI certificates is still thought of a hurdle. Full path validation with BGPsec is still a far-fetched end-goal. What we want to achieve instead is to design and implement a prototype of an inter-domain system for the management of IP prefix allocations (prefix-to-AS assignments) and BGP updates (announced/withdrawn paths-to-prefixes) that is distributed**,** peer-to-peer, tamper-resistant and independent from a central administrative authority. We need an online system that validates both IP prefix allocations and BGP updates; since we cannot embed it in inter-domain routing directly for several reasons, we consider the blockchain technology for building and maintaining a transaction ledger that will be public and allow peers to verify path advertisements and IP prefix assignments while also being secure. Our system will operate in parallel with BGP routing and will be used to detect invalid advertisements passively.

1.2. Thesis Assignment / Tasks

The tasks that were assigned to me for this thesis are the following:

1. Study the principles of blockchain technology, as well as enabling practical mechanisms.
2. Study the basics of BGP routing mechanisms, as well as current security solutions in the field of inter-domain routing (e.g., RPKI for route origin authentication, BGPSec for AS-path validation).
3. Study a state-of-the-art proposal on the concept of an "Internet Blockchain" [3], as an alternative solution to RPKI [6] and BGPSec [7].
4. Design and implement a minimal viable prototype framework for a blockchain-based solution for the management of IP prefix allocations and BGP updates, that works in parallel with everyday routing operations, validating them online.
5. Evaluate the framework using real BGP updates, in terms of scalability, efficiency and other potential metrics.

1.3. Thesis Structure

The structure of this report is the following. First, we analyze the background of the thesis and related work (Section 2). That includes studying the BGP protocol and how Internet routing works, the basics of the blockchain technology, the proposed Internet Blockchain paper and how it could help in making the Internet more secure. In the System section (Section 3), we analyze our own framework. We detail what our chain is, the blocks which the chain consists of and the types of transactions that are included in these blocks. We also go in depth into how our distributed blockchain network operates. In particular, we analyze the interactions between the blockchain (BC) nodes, how they validate the blockchain and how they build the IP prefix allocation state and AS-level graph state based on the valid transactions in the chain; keeping the correct states plays a key role in the validation of the transactions. Following the System section, we evaluate our setup using emulation, we describe what checks we perform in order to make sure everything is correct

(as well as other tests we run for ensuring correctness), and discuss about relevant speed metrics, as well as about how scalable our blockchain could possibly be (Section 4). Finally, we conclude the thesis report and describe directions for future work (Section 5).

2. Background and related work

2.1. BGP and Internet Routing

The Border Gateway Protocol (BGP) is a standardized distributed gateway protocol which is designed to exchange routing and IP prefix reachability information among Autonomous Systems (AS) in the Internet, subject to their -local- preferences and export policies. An AS is a network of variable size, which for simplicity can be treated as a set of routers and connecting links, run by a single organization. All those networks (ASes) combined make up the entire Internet. BGP is a destination-based Path Vector Protocol, which maintains paths to different networks and gateway routers and determines the routing decisions based on that knowledge and a prioritized sequence of rules used for breaking ties among routes and selecting the best path to a destination prefix. In a nutshell, it keeps the Internet running. BGP is like the postal service of the Internet [4]. It is responsible for looking at all of the available paths the data could travel and picking the best route, which usually means hopping between autonomous systems. BGP makes routing decisions based on paths, network policies, or rule-sets configured by a network administrator and is involved in making core routing decisions.

By default, routers running BGP advertise to and accept advertised routes from other BGP routers, in the form of BGP updates. BGP updates can be either advertisements or withdrawals, while there is a number of protocol-specific messages to establish communication which we do not mention here for brevity. ASes implicitly trust the routes that are shared with them. This enables the routing of traffic across the Internet to be automatic and decentralized but it also leaves the Internet vulnerable to attacks or accidental disruption, known as BGP Hijacking. BGP Hijacking is defined as the illegitimate takeover of groups of IP addresses by corrupting the routing tables that are managed by BGP (e.g., via advertising illegitimate sub-prefixes or paths to prefixes). This results in Internet traffic to be sent somewhere else and not where it was intended to. The attacks can be malicious or sometimes just accidents.

Proposals intending to make BGP more secure such as RPKI or BGPsec are hard to deploy due to a number of reasons [8, 25]. One of the main reasons is that BGP is embedded in the core systems of the Internet and thus fundamental changes (e.g., for drastically securing the protocol, like BGPsec) would require the coordination of tens of thousands of organizations and potentially result in a temporary shutdown of large parts of the Internet.

2.2. Blockchain

2.2.1. Brief History and Overview

A blockchain is a growing connected list of records, called blocks. All blocks are linked using cryptographic primitives. Each block must contain a cryptographic hash of the previous block, a timestamp and some data, usually referred to as transactions. It is the best known example of a

distributed ledger because, by design, it is very resistant to the modification of its data; tampering with a block “breaks” the chain.

The Blockchain was invented by Satoshi Nakamoto in 2008 [1], although the first work on a cryptographically secured chain of blocks was described by Stuart Haber and W. Scott Stornetta in 1991 [2]. Nakamoto improved the initial design in an important way, by using the proof of work system as a way to add new blocks in the chain without the need for them to be signed by a trusted authority. The design was implemented the following year by Nakamoto as the core component of the cryptographic currency Bitcoin, where it serves as a public ledger for all related transactions.

2.2.2. Proof of Work

In order to implement a distributed ledger, we can use a proof of work system. The proof of work system involves a scanning of a value, called nonce, which when hashed, will give a result with an amount of leading zeroes, as described in the Bitcoin-Blockchain whitepaper [1]. The average work required to find this value is exponential w.r.t. the amount of the leading zero bits but it is easy to verify it with a single hash. Therefore, generating it is hard and resource-intensive, while validating it is trivial; this observation is the heart of the blockchain functionality.

The proof of work essentially is forcing the nodes (the so-called “miners”) in the network to put in the CPU power needed into creating a block that cannot be done again without redoing all the work. As later blocks are chained after it, the work to change this block would require to change all the other blocks that come after it. If a majority of CPU power is controlled by honest nodes, the honest chain will always grow faster and outpace any malicious competing ones. To modify a block, an attacker would have to redo the proof of work of this block and also the one of all the following blocks after it to catch up and surpass the current valid chain. It is shown that the probability of a slower attacker catching up with the chain diminishes exponentially as more blocks are being added to the chain [1].

2.2.3. Genesis Blocks and incentives

A genesis block is the first block of the blockchain. It is a special block that does not reference a previous block. In most cases, it is hardcoded into the software of the application. It contains a timestamp and a hash that will be used for blocks that come after it but the transaction data in it should be accepted from everyone in the network.

The genesis block creates incentives for everyone to support the network and to start mining their own blocks. Specifically, in the case of Bitcoin, the genesis block creates a new coin that is owned by the creator of the block and it provides a way to start circulating coins into the network since there is no official authority that issues them (e.g., a “bank”).

2.2.4. Network

In the general case of a network that operates a blockchain, it should be run as follows, according to the original bitcoin-blockchain whitepaper [1]:

a) New transactions are broadcasted to all the nodes.
b) Each node collects new transactions into a block.

c) Each node works on finding a difficult proof of work for its block.
d) When a node finds a proof of work it broadcasts the block to all nodes.
e) Nodes accept the block only if all transactions in it are valid and not already spent.
f) Nodes express their acceptance of the block by working on creating the next block in the chain, using the hash of the accepted block as the previous hash.

The nodes in the network consider the longest chain to be the correct one and will keep working on expanding it. If there are conflicts between two nodes and they broadcast a block simultaneously, some nodes will receive one version of the chain and others the other. Eventually, however, the tie will be broken and the chain will get replaced when the next proof of work is found and the chain becomes longer.

2.3. The Internet Blockchain

The Internet infrastructure is designed to operate amongst different autonomous systems and be administered via some cooperating authorities and Internet registries. Because of the current surge in cyber attacks on the Internet, these entities might be compromised. Attacks on the core infrastructure of the Internet might result in a shutdown of parts of the Internet or cut countries or continents off from it. One example is when a Pakistani ISP accidentally started an unauthorized announcement of the prefix 208.65.153.0/24 in 2008. This resulted in YouTube being inaccessible for several hours [9]. More recently, Google took half of Japan off the Internet for two hours when 135,000 prefixes on the Google-Verizon path were announced when they shouldn't have been [14].

Many claim that the Internet needs a distributed, tamper-resistant, peer-to-peer infrastructural resource management mechanism outside the control of any single entity [3]. The blockchain is the best known mechanism for building a secure, distributed transaction ledger amongst untrusted peers which was pioneered by the creators of the Bitcoin cryptocurrency [1]. A blockchain transaction involves a resource transfer from one or more inputs to one or more outputs. In general, such a transaction provides a secure resource transfer mechanism to move resources from one or more entities to one or more entities.

We built our own prototype version of the Internet Blockchain as proposed in the Internet Blockchain paper [3] with a few differences. First, our transactions are slightly different than the ones proposed in the paper. The differences are in how we manage the input data and how we display the output of the transaction. Second, for simplicity, we do not support the multi signing of the transactions by 2-3 nodes as proposed in the paper. A transaction is only signed once using the private key of the node that made it. Finally, we only focus on IP prefixes and BGP updates as the resources we would like to protect and not DNS names, but this could be something we implement as future work. Most importantly, we move from the Internet Blockchain concept to actual implementation in the context of inter-domain routing operations.

We note that securing inter-domain routing resources (such as IP prefixes, ASNs) via blockchains has been also approached in the following works: IPChain [26] and BGPcoin [27]. However, IPChain stores only IP prefix allocation and delegation data in a 1GB Proof of Stake blockchain (holding ~150K prefixes), in contrast to our work where we store IP prefix allocation data as well as BGP updates in a Proof of Work blockchain. BGPcoin maintains a set of smart contracts in an Ethereum prototype blockchain, introducing the concept of “BGPcoin”, albeit focusing again only on validating IP prefix and ASN allocations for the purpose of origin authentication against IP prefix

hijacking.

3. System

The objective of our systematic methodology is to be able to validate, store and retrieve transactions related to inter-domain routing functions, such as IP prefix allocations and BGP updates, in a distributed, tamper-proof ledger. Our system is designed to operate in parallel with BGP routing, and is useful for detecting, for example, invalid advertisements (paths, prefixes, etc.) passively and in an online fashion.

We base our approach on the Blockchain primitives which offer the capability for forming such a ledger, and we construct our design to be able to support the BGP-related transactions required.

To understand what an AS needs to be able to do in the context of inter-domain routing, we use the following indicative workflow as an example.

1. An IP prefix registrar assigns a prefix to ASX with a specific lease period.
2. ASX leases this prefix to organization Y, to be used by ASW and ASZ.
3. ASW and ASZ advertise this prefix to their neighbors, which in turn propagate this announcement to their own neighbors.
4. ASW and ASZ withdraw the prefix since their lease period is about to expire.
5. ASX revokes the prefix allocation and retrieves ownership of the prefix.

Therefore, we want to model these different actions as BC transactions.
Our system is composed of the following components: the (Block)chain (BC), the nodes that participate in the chain formation and validation, the blocks that form the chain and the transactions that are stored within the blocks. We next analyze those components in detail.

3.1. Chain

The chain is the main data structure. It is a List (implemented as a Python collection) which simply consists of blocks (each block is a Block object as described below) that are being mined by the nodes in the BC network. The first block to be included in the chain is called the Genesis Block; this does not need to be mined, since the data in this block come from reliable sources and are used to “bootstrap” the chain. Thus, every chain starts with one block that contains one transaction for the initial IP prefix allocations (note that valid allocations of IP prefixes to ASes are prerequisite for these prefixes to start getting announced/advertised over inter-domain paths).

3.2. (BC) Network

Our network, or as we call it, the Blockchain network (BC network), is a peer-to-peer network where each peer represents an AS. Each peer or node makes transactions, can create new blocks and interacts with other nodes in the network. A node is considered to be a part of the network when it generates its own public-private key pair and broadcasts its public key to a set of nodes that are being used as "bootstrap" nodes. Nodes can leave at will and come back to (i.e., join) the network. Nodes that come back must accept the longest chain as a proof of what happened while

they were gone.

The -Internet- Blockchain network runs as follows, in contrast to the bitcoin blockchain network [1]:

a) A set of nodes called bootstrap nodes start the chain first.
b) The bootstrap nodes discover each other and become peers.
c) Any new node that wants to join should broadcast its public key to them.
d) Each node broadcasts the transactions it makes.
e) Each node collects incoming transactions that will be added to a new block if it chooses to create (i.e., mine) one.
f) The nodes that want to create a new block must find a proof of work for this block.
g) Once a node finds a proof of work it adds this block to its chain and broadcasts a message so that other nodes will resolve the conflicts w.r.t. the “correct” chain.
h) Other nodes accept the new chain only if the transactions in it are valid and the new chain is longer than theirs.

Some miscellaneous requests the nodes in the network accept, as well as their return values are presented below:

Chain: Returns the chain.
Request Method: GET
Request URL: 'http://node_ip:node_port/chain'
Return Values:

Fields	Type	Description
chain	String	The blockchain
length	Int	The length of the chain i.e., number of blocks

Topologies: Returns the AS-level graphs for all IP prefixes.
Request Method: GET
Request URL: 'http://node_ip:node_port/topos'
Return Values:

Fields	Type	Description
each prefix	Dictionary	A dictionary with the prefixes as keys and the AS-level topology (graph) as value.

Graph Visualization: Creates an image file of the graph for a specific prefix.
Request Method: POST
Request URL: 'http://node_ip:node_port/gv

Request Parameters:

	Name	Type	Description
Required	prefix	String	The prefix whose graph we want.

Find transaction by id:
Searches for a transaction given a transaction id (txid)
Returns the transaction if it is in the chain.
Request Method: POST
Request URL: 'http://node_ip:node_port/transactions/find_by_txid'
Request Parameters:

	Name	Type	Description
Required	txid	String	A valid BC transaction id.

3.3. Blocks

A block in the blockchain is an instance of the Block class. It includes the following properties:

1. **Timestamp**: Unix timestamp of when the block was created.
2. **Mined timestamp:** Unix timestamp of when the block was mined.
3. **Transactions**: A list of all the transactions included in this block (can be composed of more than 1 elements).
4. **Previous Hash**: The hash of the previous block in the chain (String).
5. **Nonce**: A value for verifying the proof of work (hash) of this block.
6. **Index**: The position of the block in the chain.
7. **Hash**: A SHA-256 hash with an amount of leading zeros that determine the difficulty of the proof of work.
8. **Signature**: A digital signature signed by the miner.
9. **Miner**: The node (AS) that mined this block.

The above properties are being set when the block is mined and can all be found in the chain.

3.4. Transactions

A transaction is one of the main elements that make up the blockchain. Each transaction is a collection of data that we need to evaluate before including them in the blockchain (i.e., mine them in blocks). All transactions occur between peers and there is not a central authority or registry involved.

There are two main categories of transactions. They are modeled as Python Classes: one is called IP Address Allocation and the other is called BGP Update. IP Address Allocation is being subclassed by the Assign, Revoke and Update classes and BGP Update is being subclassed by BGP_Announce and BGP_Withdraw.

These transactions are the ones that are being made and broadcasted by the nodes in the network and thus end up in blocks in the chain.

There is also a special type of transaction called *Genesis Assign* which it is a bootstrap transaction to validate initial IP allocations. This transaction is the first transaction in the chain and it is included in the genesis block when a node runs the main BC script for the first time.

In the following, we detail how a transaction of any type looks like in the chain. The code below shows it in JSON format. The transactions in a block are stored in an array that includes multiple transactions. All transaction data (such as input, output, etc) are described below and vary for each transaction type.

```
"transactions": [
   {
     "signature": [a really long number],
     "trans": {
       "input": [varies for each transaction type],
       "output": [varies for each transaction type],
       "timestamp": the time the transaction was made,
       "txid": the transaction id,
       "type": varies for each transaction type
      }
   }, ...
 ]
```

And, thus, for each transaction type we have the following context-specific details.

Genesis Assign transaction

The first transaction to be included in the chain. It retrieves IP allocation data from trustworthy sources and formats them in a specific output format.
This transaction in the chain includes the following:

- **Input**:
  Data from source being formatted as a list containing multiple lists, that include the prefix and the AS number(s) that this prefix has been assigned to, e.g.
  ```
  [
      [  Prefix X1 , ASX1  ]
            ,...,
      [  Prefix XN, ASXN  ]
  ]
  ```
- **Output**:
  Data from source being formatted as a list containing multiple lists, that include the prefix and the AS number(s) that this prefix has been assigned to, e.g.
  ```
  [
      [  Prefix X1 , ASX1  ]
            ,...,
     ,[  Prefix XN, ASXN  ]
  ]
  ```

- **Timestamp**:
  A UNIX timestamp of when this transaction was created.
- **Txid**:
  A unique identifier of the transaction. In the case of the genesis assign transaction this is set to -1.

IP Allocation - Assign

This type of transaction is made when an AS wants to assign the prefix it currently owns to one or more ASes for a duration of time (lease, in months). The original owner of this prefix loses any claim over it until the lease duration has expired. An AS that makes this transaction must provide the following parameters when making the request.

Request Method: POST
Request URL: 'http://node_ip:node_port/transactions/assign/new'
Request Parameters

	Name	Type	Description
Required	prefix	String	The prefix to be assigned.
Required	as_source	String	The AS that makes this transaction.
Required	as_dest	Array	A list of the ASes that the current owner wants to assign the prefix to.
Required	source_lease	Int	The lease duration as it was assigned to the owner making this transaction.
Required	lease_duration	Int	How long this prefix will be assigned to the destination ASes. source_lease must be greater or equal than this value.
Required	transfer_tag	Boolean	Describes whether the new owners of this prefix will be able to assign it further to other ASes as well.
Required	last_assign	String/Int	The transaction id (txid) of the assign transaction the current owner received this prefix from. (If this is the genesis transaction just provide the int -1).

This transaction in the chain includes the following:

- **Input:**
  A list of the request parameters formatted as:

```
[
    prefix,
    as_source,
    [ ..., as_dest[i], ... ],
    source_lease,
```

```
    lease_duration,
    transfer_tag,
    last_assign
]
```

- **Output:**
  A list that contains lists for every AS in the as_dest list from the request parameters.
  ```
  [
   ...,
   [
     prefix,
     as_dest[i],
     lease_duration,
     transfer_tag
   ],
   ...
  ]
  ```
  where `0 ≤ i < N`, `N` the number of ASes in the as_dest list.
- **Type:**
  The type of the transaction. In this case, the type is "Assign".
- **Timestamp:**
  A UNIX timestamp of when this transaction was made.
- **Txid:**
  A unique identifier of the transaction. A SHA-256 hash of the parameters as_source, last_assign and the time the transaction was made.
- **Signature**
  A digital signature, created with as_source's (the AS that makes this transaction) private key so that it can be verified by anyone in the network using as_source's public key.

IP Allocation - Revoke

This type of transaction is made when an AS wants to revoke the prefix from all the ASes it had previously assigned it to. This transaction is only valid once the assigned lease duration has actually expired. Then the new lease is being calculated as `new_lease = as_source_lease - assigned_lease`. The AS that makes this transactions claims back the prefix for a new_lease period of time. We've also implemented an algorithm that checks periodically the IP Allocation - Assign transactions to see if a lease has expired and automatically makes a new IP Allocation - Revoke transaction.

An AS that makes this transaction must provide the following parameters when making the request to the service.

Request Method: POST
Request URL: 'http://node_ip:node_port/transactions/revoke/new'
Request Parameters:

	Name	Type	Description
Required	as_source	String	The AS that makes this transaction.

Required	assign_tran	String	The transaction id (txid) of the Assign transaction we want to revoke.

This transaction in the chain includes the following:

- **Input:**
  A list of the request parameters formatted as:
  `[ as_source, assign_tran ]`
- **Output:**
  A list that includes the previously assigned prefix found in the assign transaction given as a parameter, the as_source, the new lease duration for the owner that takes back the prefix, and the initial transfer tag (this will always be True).
  ```
  [
      prefix,
      as_source,
      new_leaseDuration,
      transfer_tag
  ]
  ```
- **Type:**
  The type of the transaction. In this case, the type is "Revoke".
- **Timestamp:**
  A UNIX timestamp of when this transaction was made.
- **Txid:**
  A unique identifier of the transaction. A SHA-256 hash of the parameters as_source, assign_tran and the time the transaction was made.
- **Signature:**
  A digital signature, created with as_source's (the AS that makes this transaction) private key so that it can be verified by anyone in the network using as_source's public key.

IP Allocation - Update

This type of transaction is made when an AS wants to update the lease period of a prefix it had previously assigned to another AS. The lease should not have expired and the AS making the transaction cannot exceed the lease duration it was given at genesis or at any other assign transaction.

An AS that makes this transaction must provide the following parameters when making the request to the service.

Request Method: POST
Request URL: 'http://node_ip:node_port/transactions/update/new'
Request Parameters:

	Name	Type	Description
Required	as_source	String	The AS that makes this transaction.
Required	assign_tran	String	The transaction id (txid) of the Assign transaction the as_source wants to

			revoke.
Required	new_lease	Int	The new lease period in months.

This transaction in the chain includes the following:

- **Input:**
  A list that includes the parameters from the request as:
  `[ as_source, assign_tran, new_lease ]`
- **Output:**
  A list that contains lists for every AS in the as_dest array found in the assign transaction whose id was given as a request parameter. Those lists include the prefix, the AS and the transfer tag all found in the assign transaction and the new lease given as a parameter in this transaction, formatted as

```
[
 ...,
 [
   prefix,
   as_dest[i],
   new_lease,
   transfer_tag
 ],
 ...
]
```

  where $0 \leq i < N$, N the number of ASes in the as_dest array from the Assign transaction.
- **Type:**
  The type of the transaction. In this case, the type is "Update".
- **Timestamp:**
  A UNIX timestamp of when this transaction was made.
- **Txid:**
  A unique identifier of the transaction. A SHA-256 hash of the parameters as_source, assign_tran and the time the transaction was made.
- **Signature:**
  A digital signature, created with as_source's (the AS that makes this transaction) private key so that it can be verified by anyone in the network using as_source's public key.

BGP Update - Announce

This type of transaction is made when an AS wants to advertise a prefix it learned from some -neighbor- ASes to some other -neighbor- ASes.

An AS that makes this transaction must provide the following parameters when making the request.

Request Method: POST
Request URL: 'http://node_ip:node_port/transactions/bgp_announce/new'
Request Parameters:

	Name	Type	Description
Required	prefix	String	The prefix to be advertised/announced.
Required	as_source	String	The advertising AS.
Required	as_source_list	Array	ASNs from where as_source learns the prefix (Array of strings).
Required	as_dest_list	Array	ASNs to which as_source advertises the prefix (Array of strings).
Required	bgp_timestamp	String	Timestamp of learned BGP update.

This transaction in the chain includes the following:

- **Input:**
  A list that includes the request parameters formatted as,
  ```
  [
    prefix,
    as_source,
    as_source_list,
    as_dest_list,
    bgp_timestamp,
  ]
  ```
- **Output:**
  A list that contains all the paths for this prefix, between the as_source, every AS in as_source_list and all the ASes in as_dest_list.
  (e.g. for prefix = X, as_source = ASY, as_source_list = [AS1, AS2], as_dest_list = [AS3, AS4] we have the following paths:
  X-AS1-ASY-AS3
  X-AS1-ASY-AS4
  X-AS2-ASY-AS3
  X-AS2-ASY-AS4)
  ```
  [
   ...,
   [
     prefix,
     as_source_list[i],
     as_source,
     as_dest_list[0...M-1]*
   ],
   ...
  ]
  ```
  where $0 \leq i < N$, N the number of ASes in the as_source_list and M the number of ASes in the as_dest_list.
  * Every element of as_dest_list, for every i in as_source_list.

- **Type:**
  The type of the transaction. In this case, the type is "Announce".
- **Timestamp:**
  A UNIX timestamp of when this transaction was made.
- **Txid:**
  A unique identifier of the transaction. A SHA-256 hash of the parameters prefix, as_source and the time the transaction was made.
- **Signature:**
  A digital signature, created with as_source's (the AS that makes this transaction) private key so that it can be verified by anyone in the network using as_source's public key.

BGP Update - Withdraw

This type of transaction is made when an AS wants to withdraw a specific prefix from all the other ASes it had previously advertised it to. A node that makes this transaction must provide the following parameters when making the request.

Request Method: POST
Request URL: 'http://node_ip:node_port/transactions/bgp_withdraw/new'
Request Parameters:

	Name	Type	Description
Required	prefix	String	The prefix to be withdrawn.
Required	as_source	String	The withdrawing AS.

This transaction in the chain includes the following:

- **Input:**
  A list that contains the request parameters prefix and as_source.
  `[ prefix, as_source ]`
- **Output:**
  A list that includes the prefix and the withdrawing AS
  `[ prefix, AS ]`
- **Type:**
  The type of the transaction. In this case, the type is "Withdraw".
- **Timestamp:**
  A UNIX timestamp of when this transaction was made.
- **Txid:**
  A unique identifier of the transaction. A SHA-256 hash of the parameters prefix, as_source and the time the transaction was made.
- **Signature:**
  A digital signature, created with as_source's (the AS that makes this transaction) private key so that it can be verified by anyone in the network using as_source's public key.

3.5. BC node interactions

3.5.1. Network bootstrapping

At the beginning of the BC setup phase, there is a number of nodes provided in a file (bgp_network.csv) that contains the IP Address, Port and AS number of the nodes that will run the main BC script first. These nodes are called bootstrap nodes and their task is to run the main script first so that other nodes that want to participate are able to discover the network through them. The bootstrapping happens offline before a new node has discovered any other nodes.

3.5.2. Node discovery

When a node enters the network, it first registers the bootstrap nodes as neighbors. Because of this, the bootstrap nodes have a full view of the network at any given time. Every node that will ever join the network will register these bootstrap nodes first and the bootstrap nodes will register any node that contacts them as well. Considering, now, that at least one of the bootstrap nodes is up, the node that wants to enter the network will send a request to the bootstrap nodes to find out who their neighbors are, so that all of the neighbors of the bootstrap nodes (that would, essentially, be the rest of the network) and the requesting node can register each other. And so the requesting node has now successfully discovered the rest of the BC network.

3.5.3. Public node key exchange

After discovering the rest of the network, a node must broadcast its public key to all the neighbors it just registered. The receiving nodes will then update the entry with the public key in their dictionary where they keep all of information about their neighbors. Any neighbor will also register the node if this is the first time they have communicated with each other.

Now there is one more thing to be done, and that is for the node to request the public keys of all the nodes it just sent its own public key to. The node, then, updates the entries of all the nodes that replied with their public key and the exchange is over. When one of a node's neighbors has not contacted the node in a really long time (see following section), then their entry and public key get deleted and they have to be re-registered should the neighbor come back online.

3.5.4. Node liveness checks

When a node joins the network, it sends an "alive" message to all of its neighbors. Each node manages an alive neighbors dictionary with key the url of the neighbors and value the time of the received alive message. Each node updates then the appropriate entry every time one neighbor sends an alive message. Once a node sends the first alive message, a timer starts with a callback function that sends an alive message to all the neighbors every 20 seconds.

Neighbors are considered "dead" if they haven't sent an alive message in more than 60 = 3x20 seconds. A timer with a callback is started that checks the alive neighbors dictionary every 60 seconds. This callback checks every neighbor as follows:

```
time_passed = time_now - alive_neighbors[neighbor]
```

If time_passed > 60 seconds then it removes the neighbor from the dictionary and from the registered nodes list, otherwise, it does nothing (i.e., the neighbor is still alive).

3.5.5. Broadcasting transactions

Every new transaction created must be broadcasted to all the nodes in the network. By doing so, it allows for every node to validate incoming transactions and reject possibly malicious ones. Even if the node that created a malicious transaction is a trusted node the transaction will be verified by other nodes as well. The receiving nodes will be able to perform checks for its validity based on the states they hold for each prefix and thus reject each invalid transaction. If the malicious node creates invalid transactions and mines them into blocks, then the other nodes will not accept them and thus reject the chain that contains these invalid transactions the malicious node made. Another benefit for broadcasting transactions, also, is that if a transaction is valid, then it can be mined later by other nodes in the network and not necessarily by the one who created it, thus, leaving the mining process to more powerful machines, if needed.

3.5.6. Mining

Each miner node (in our setup, any node can become a miner) collects all the transactions it makes itself, together with the incoming ones from the BC network, and temporarily stores them into a list; called pending transactions. Then a new block can be created so that it includes all these transactions. The process of creating a new block that includes all the new transactions is called mining. This process is time consuming and requires a lot of computational power so that it is impossible for a malicious user to go back and tamper with all the previous blocks in the chain.

Every node can mine new blocks in the network. There is no limit to what a node can mine. It can mine blocks with self-made transactions or transactions that were made by other nodes in the network making it possible to have exclusive miner nodes.

Before mining a new block, we must make sure that we have the most up to date chain and that the transactions we are about to include in the new block, don't already exist in a block in the chain. If we don't have the most up to date version of the chain, we must request it from the network and then remove every pending transaction that is already in the chain we just received. After that, we create a new block and run the proof of work algorithm. The algorithm searches for a value that produces a certain string and once it terminates, the miner must sign the new block and add it to the chain. Both the IP Allocation state and the graph for each prefix get updated based on the type of transactions that were just mined, and finally, the miner node sends a message to all the neighbors in the network to let them know that the chain has just been updated with one new block. After successfully mining a block, the miner needs to then remove all the pending transactions that are now in the chain.

3.5.7. Longest chain resolution

As each node mines new blocks, they add these newly mined blocks in the chain. And because the nodes mine independently, it is possible that there could be more than one versions of the chain in the network at a given time. So there must be an algorithm on which we need to rely so that every node in the network has the most recent and correct version of the chain. This algorithm is a consensus algorithm that we use in order to resolve conflicts.

The steps for resolving the conflicts are the following:

1. **Request all the chains in the network:**
   First ask all of your neighbors to give you their version of the chain. If a neighbor replied proceed to step 2, otherwise, continue with the next neighbor and so on until you have tried to contact all of your neighbors.
2. **Compare each chain with your own:**
   Check the length of every chain you receive. If the length of one of the incoming chains is greater than your chain then check if the chain is also valid. If the chain passes the validity checks then go to step 3. Otherwise reject the chain and continue with the remaining neighbors from step 1.
3. **Replace the chain:**
   If you have discovered a new valid chain then replace your old chain with this one. The old chain is now outdated and should be replaced. Update the state based on the new transactions and then go back to step 1 and check the remaining chains.

3.5.8. Validating the BC online

3.5.8.1. Validating overall BC consistency

There are various checks in place for validating the blockchain's consistency overall. The checks listed below happen every time a node receives a new chain from the network.

First of all, for each block, we check that the hash of the previous block stated in the current block is indeed the same as with the previous one in the chain. This is easy to verify and is the basic principle of any blockchain according to the Blockchain Whitepaper [1].

One easy thing to verify, also, is the proof of work of any block. Each block contains a value (nonce) that was used towards creating a hash with a set amount of leading zeroes. The number of leading zeroes determines the difficulty of the proof of work (i.e., the miner needs to spend more computational power to find the value (nonce) that produces this hash). Once this value is known, it is very easy to verify it and, thus, make sure that the miner spent a lot of power to mine this block.

Lastly, we verify the origin of the miner of each block. Each miner signs a block with its private key and thus, another node in the network can verify the origin of the miner by using the miner's public key.

3.5.8.2. Checking transactions before mining

The IP Allocation transactions are checked with the help of a dictionary that holds information about the state of each prefix (see also Section 3.6). This dictionary uses each prefix as a key and the value is a list that includes multiple tuples which include the following:

```
(ASN, lease duration(in months), transfer tag, txid),
```

where ASN is the AS Number, lease duration is for how long an AS owns the prefix, the transfer tag indicates if the current owner (ASN) can assign the prefix to other ASes and the txid is the id of the last valid IP Allocation - Assign transaction (i.e., how ASN got the prefix).

For the BGP Update transactions we use a directed graph as a state to track every announce/withdraw transaction per prefix (see also Section 3.7). The nodes in this graph represent ASes and the edges show the direction of the announcements. The first node in the graph is the prefix and the first edge is between the prefix and the owner of this prefix based on the IP Allocation state. The direction of the edges is always towards the prefix, so, for example, for prefix X if owner ASY announces the prefix to ASW the graph would look like: X ← ASY ← ASW.

The first check we perform for every type of transaction is the verification of the author's identity. By verifying its digital signature we can see if the node that made this transaction is in the BC network. Only then we continue to perform further checks on the transaction. If we cannot verify the node's origin then the transaction is not accepted.

The validity checks for the IP Allocation type of transactions are as follows:

IP Allocation - Assign

First, we need to check whether the AS source is able to assign this prefix to the destination ASes. An AS can assign the prefix only if (i) it can provide a transaction id of the last valid assignment (i.e. how it got the prefix), (ii) its own lease duration is greater (or equal) than the one it is assigning to other ASes, and (iii) it has the right to transfer the prefix to others. We also check if the destination ASes are in the blockchain network or not. Any transaction that fails one of the the above checks is considered invalid and it is not accepted.

IP Allocation - Revoke

We have to make sure that the lease has actually expired for this transaction to be valid. If it has expired, we check whether the node that makes this Revoke transaction is also the one that made the Assign transaction whose id is provided as a parameter for this transaction. We must also check if the destination ASes in the Assign transaction own this prefix currently. If any of the above checks fail, the transaction is considered invalid.

IP Allocation - Update

For the Update transactions, we want the lease of the current owners of the prefix to not have expired. We must then also check if the AS that makes this Update transaction is the same as the one that made the Assign transaction given as a parameter. Finally, we check if the destination ASes in the Assign transaction provided, currently own the prefix. If any of the above checks fail, the transaction is considered invalid.

And for the BGP Updates type of transactions, we have:

BGP Update - Announce

First, we verify the origin of this transaction based on the topology of this prefix. We then check if all the ASes in as_source_list and as_dest_list are in the blockchain network. We only accept transactions that include ASes that are in the BC network. Lastly, we check if this transaction introduces loops in the topology of this prefix. If any of the above checks fail and there are loops in

the graph after this transaction, the transaction is not accepted.

BGP Update - Withdraw

For BGP Withdraw transactions, except for the node's signature check, we also check if there is at least one path from the Withdrawing AS that leads to the prefix in the topology. If there is at least one then the Withdraw transaction is considered valid.

If a transaction passes all the above checks, it is being stored in a list called *pending transactions* along with other valid transactions that are waiting to be mined. If not, the id (a SHA-256 hash) of this transaction is stored in a list called *invalid transactions*, which helps us identify any invalid transactions made by a possibly malicious miner, as explained in the next section.

3.5.8.3. Checking transactions in mined blocks

Transactions in mined blocks should also be checked for their validity. Let's assume that a participating node mines blocks with invalid transactions in them. Then, others could potentially replace their chain with this one that contains invalid transactions. A solution to this problem is to mark all the invalid transactions as they happen. A newly made transaction must be broadcasted to all the nodes in the network so each one of them can verify whether it's valid or not. And thus, before replacing the chain, a node can look for all the invalid transactions it has marked in the incoming chain. If there is at least one invalid transaction in the new chain, the chain is rejected from the network. Nodes that leave the network and come back or new nodes that join the network accept that the transactions in the current chain are all valid and update their IP prefix allocation state and AS-level graph state accordingly.

3.6. Building the IP prefix allocation state

The IP prefix allocation state holds information about each prefix such as: which AS currently owns it, for how long, if it can transfer the prefix to other ASes and the id of the last valid IP Allocation - Assign transaction the AS got the prefix from.

At the creation of the Genesis block, the state is being instantiated from data gathered from reliable sources. Each prefix is being assigned to at least one AS for a long period of time and with the ability to transfer it to others as well.

The state gets updated after every change that happens to the chain. That includes the time when a node mines one new block or when the chain gets replaced by another one from the network. We go through every block from the beginning of the chain and update the state according to every type of transaction. For the IP Allocation - Assign transactions, we remove the current owner from the state dictionary and add all the ASes the owner is assigning the prefix to. For the IP Allocation - Revoke transactions, we remove the current owners of the prefix and restore the AS that made this transaction. For the IP Allocation - Update transactions, we just update the lease of the respective entries.

3.7. Building the AS-level graph state

The AS-level graph state consists of directed graphs for each prefix. It is a dictionary that uses each prefix as key and as value the directed graph for this prefix. We use the graphs to keep track of each BGP announcement and for validity checks.

Just like the IP Allocation state, this state is also being initiated at the creation of the Genesis block. We first create a graph for every prefix and add the first node which is the prefix. After that, we add an edge between an AS and the prefix node for every AS that owns the prefix based on the IP allocation data we gathered.

After every update that happens to the chain, this state is also being updated after every transaction that affects it.

After a new IP Allocation - Assign transaction, we must clear the previous topology for this prefix e.g. remove the AS node that is transfering the prefix and all the edges that include this node and add new nodes and edges for the ASes that are now the owners of this prefix. For the IP Allocation - Revoke transaction, we remove the nodes of the ASes that got the prefix from the Assign transaction and the respective edges, add a node for the AS that transferred the prefix, i.e the original owner node, and that is now making this revoke transaction and also add an edge between this original node and the prefix node.

After a new BGP Update - Announce transaction, we add new edges to the graph that show which AS announced the prefix to which ASes. The edges are directed towards the prefix and so edges are being added from the AS source that made this transaction towards the ASes from which it learned the prefix and from the ASes the source is advertising to towards the source.

After a BGP Update - Withdraw transaction, we remove the edges between the withdrawing node and its predecessors. We also remove all the other nodes that cannot reach the prefix now, after withdrawing these edges. For simplicity, we assume that withdrawals come earlier than revoke transactions for each prefix, meaning that the graph is already clear of stale routing information upon assignment revocation.

There is also support for the visualization of the AS-level graph as described in section 3.2.

4. Evaluation

4.1. Setup and Emulation

4.1.1. Input Dataset

For the initial IP prefix allocations during the "genesis" bootstrap period, we use data from CAIDA (Center for Applied Internet Data Analysis) [18]. The dataset contains IPv4/IPv6 Prefix-to-Autonomous System (AS) mappings derived from RouteViews data (http://www.routeviews.org).

For BGP Updates we use data gathered from BGPStream [5], an open-source software framework for live and historical BGP data analysis. The data collected from BGPStream are from RouteViews and RIS (RIPE Routing Information Service https://www.ripe.net/analyse/internet-measurements/routing-information-service-ris). For this report, we used data for the prefixes 139.91.2.0/24, 139.91.0.0/16 and 139.91.250.0/24, that are announced with the FORTH AS as origin. The bgp updates we used were collected on 25th of July 2018 from 9am to 10am. We replay these updates as BGP Announce transactions as we will discuss later.

4.1.2. AS BC Agents as Processes

We map each configured BC agent to a process running our framework. Each process has an IP and port and so it is reachable on the host machine. The process also has an AS number so that it represents an AS. These values are given when the node wants to run the framework the first time. The process connects the agent to the bootstrap nodes first, and then it discovers the rest of the network through them. After that the process broadcasts the agent's self-generated public key to the network and then requests all other nodes' public keys to store them locally. After that, to keep track of which nodes are alive and active in the network, the process starts two threads; a thread with a callback function that sends "alive" messages to the neighbors and a thread with a callback function that asks the neighbors if they are still online. After all that, the agent has entered the network and can start making transactions and mining new blocks.

4.1.3. Replaying BGP updates as transactions

With the help of a script that parses the data from BGPStream [5] (mentioned above), we make BGP Announce transactions in the network.
The bgp file contains data of this format:

```
prefix|origin AS|path|project|collector|type|timestamp|peer AS
```

So for example, a real BGP advertisement looks like this:

```
139.91.2.0/24|8522|2914,1273,3329,12361,8522,8522,8522,8522,8522,8522|ris
|rrc12|A|1532509346|2914
```

After removing the prepending ASes from the path we get:

```
[ 2914, 1273, 3329, 12361, 8522 ]
```

where 8522 is the origin of the prefix (here, 139.91.2.0/24). The direction of BGP advertisements is from right to left, while traffic towards the prefix flows from left to right.
From that path, we create and replay 4 BGP Announce transactions in the network, in the following format:

```
origin AS - advertising AS - destination AS
```

In our example, the 4 transactions would be:

```
1) 0      - 8522  - 12361*
2) 8522   - 12361 - 3329
3) 12361  - 3329  - 1273
4) 3329   - 1273  - 2914
```

At this moment, we only replay advertisements from the data we gathered (and not withdrawals), but it would be an easy feature to add withdrawal support in the future.

Note also that despite the fact that from this update, 4 transactions were generated, these 4 transactions will not need to be created again for another update which includes the same (sub-)path from 2914 to 8522. In general, if an update propagates from ASX to ASY and then to ASZ, this is equivalent to a single transaction conducted by ASY, with ASX as input and ASZ as output. Therefore, the number of transactions can actually be orders of magnitude smaller than the number of updates, which are propagated again and again during the BGP path exploration process; we encode though advertisement information on a neighbor-to-node-to-neighbor basis (instead of a full path basis). Full paths are then found (and validated) by examining the constructed graph. Smart aggregation of advertisement information (across neighbors, for example) in few transactions and blocks is the subject of future work.

*(origin 0 means that the advertising AS currently owns the prefix)

4.1.4. Evaluation environment specifications

All the tests were emulated over a commodity laptop with the following specifications:

1. **CPU cores:** 4
2. **Processor Base Frequency:** 2.60 GHz
3. **RAM:** 16GB
4. **Python version:** 3.4.3

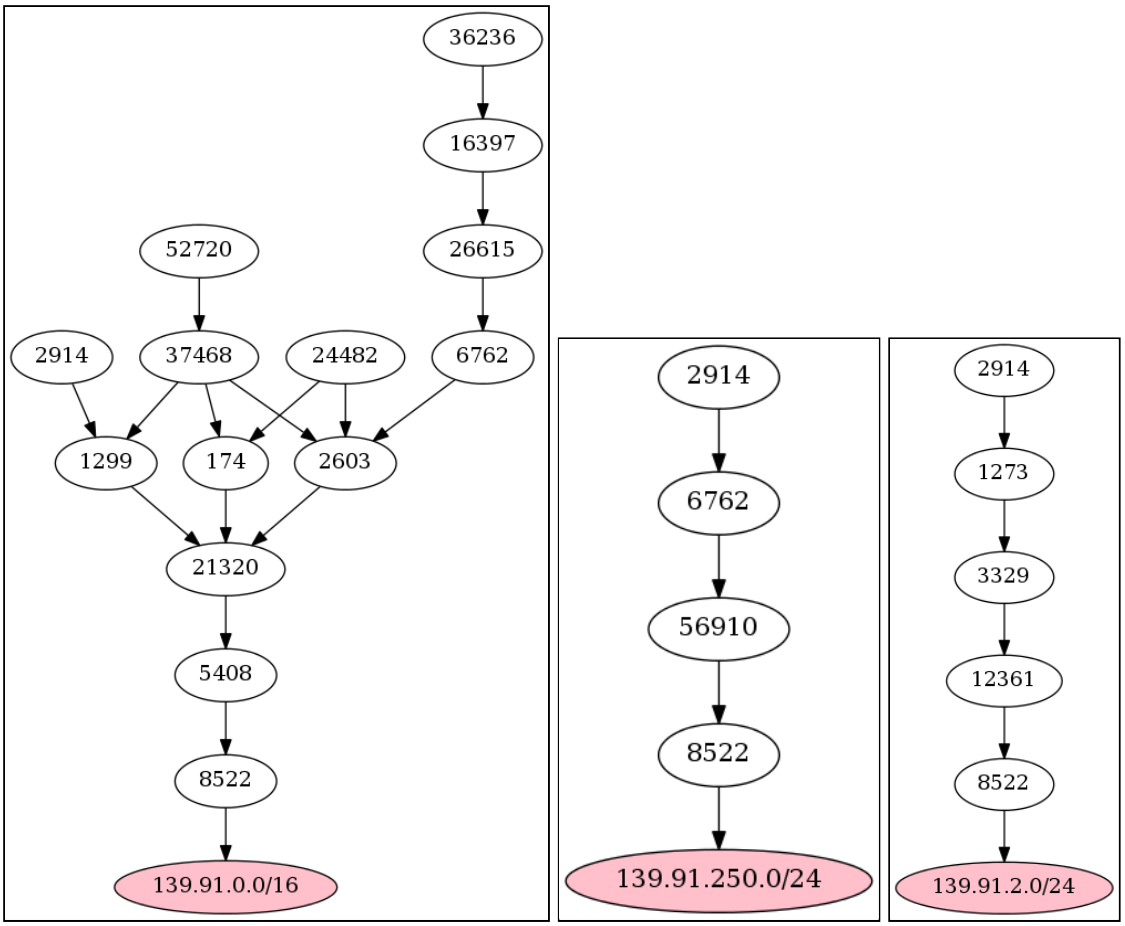
4.2. Correctness

4.2.1. Expected vs BC-generated graph

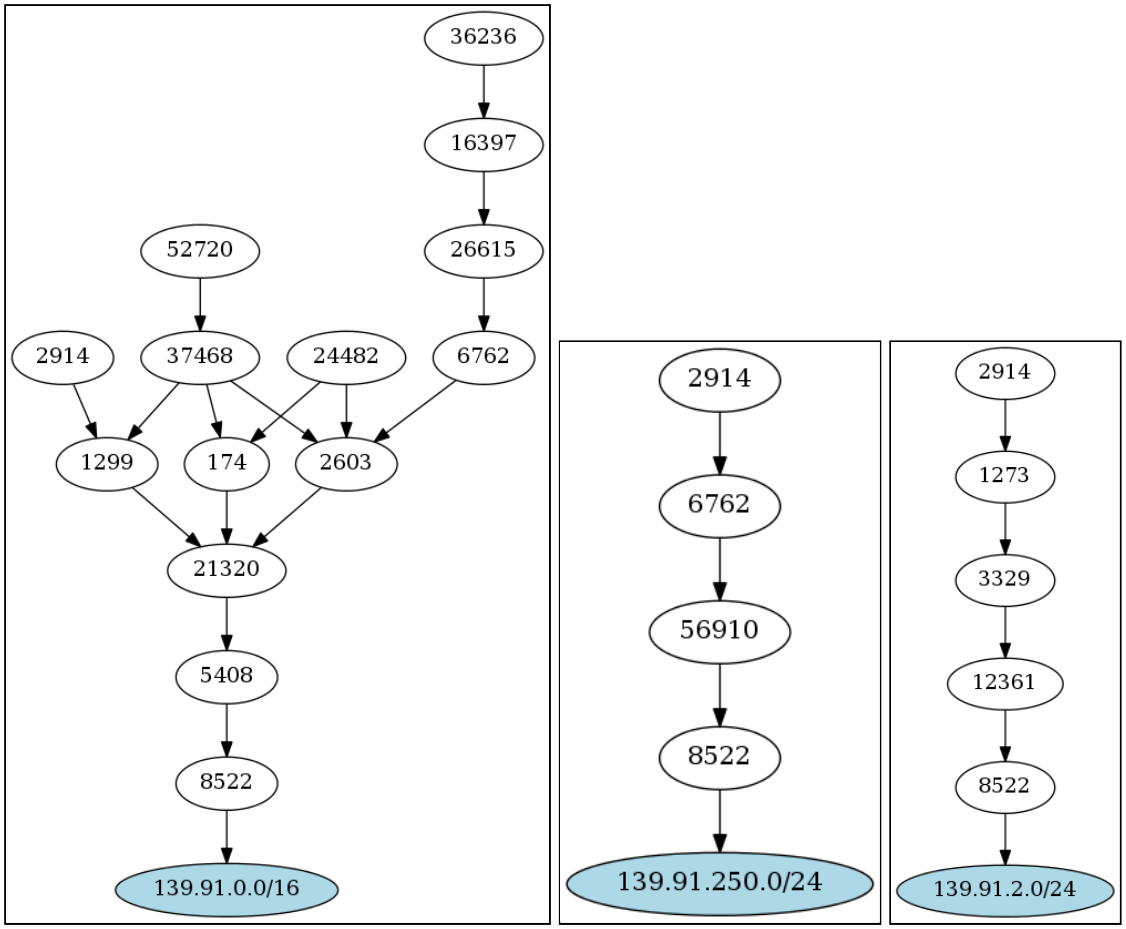
For checking the correctness of the graphs that are generated from the BC network based on the BGP transactions, we wrote a script that parses raw BGP update data and generates an expected graph for each prefix found in these updates. The graphs generated do not involve the network or the blockchain at all.

By replaying the data mentioned above as BGP Announce transactions, the BC network generates the graphs for these prefixes as well. To check that the network created the graphs just like we would expect it to, we compare each graph of every prefix from the network with the ones we generated without involving the network.

So, for example, in the BGP update file there are BGP advertisements that involve the prefixes 139.91.0.0/16, 139.91.2.0/24 and 139.91.250.0/24. Running the script that checks the correctness, gives these 3 expected graphs:



So now we need to check if the graphs of these prefixes from the network are the same as the above graphs. By requesting the images of the graphs from the BC network we get the following:



As we can see, the graphs are the same. That means the transactions were created, checked and mined successfully by the nodes in the network.

For the above checks, in addition to the visualization of the graphs we also use a correctness algorithm that compares all the generated graphs from the raw updates with the corresponding ones generated from the network and outputs the result in a console.

4.2.2. Other tests

IP Allocation: Assign - Update - Revoke transactions

In this test, we will emulate a series of transactions (as steps in an example workflow) for the prefix 1.0.0.0/24. First, we will transfer the prefix from one AS to several other ASes. After that, we will update the lease period for the ASes we transferred the prefix to and then, after the lease has expired, we will finally revoke the prefix allocation and give the prefix back to the original owner.

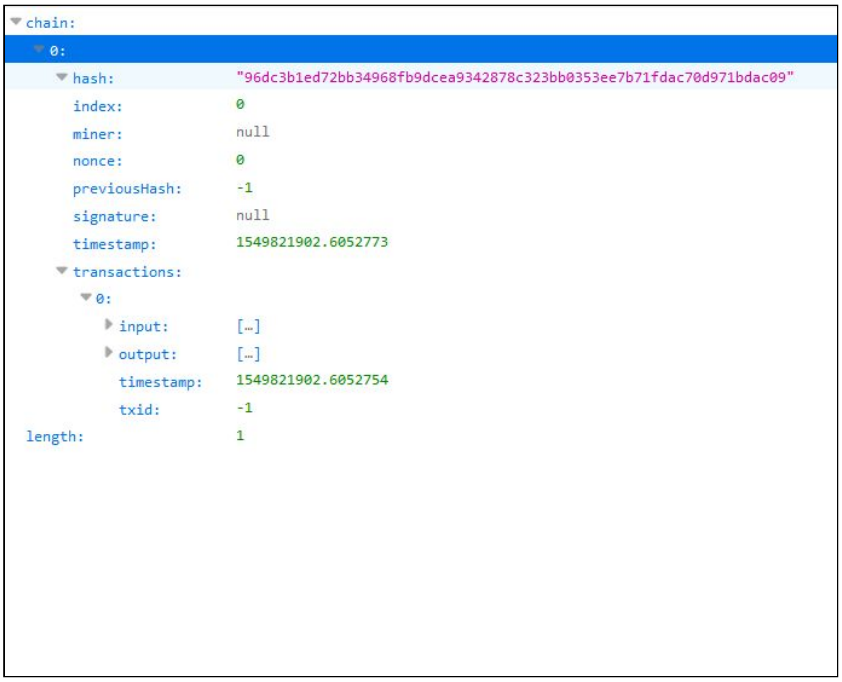
For the sake of simplicity, the network will consist only of the bootstrap nodes. Their details are as follows (the ASNs are selected randomly):

Node # *	IP	PORT	ASN
0	localhost	5000	13335
1	localhost	5001	133741
2	localhost	5002	133948
3	localhost	5003	133955
4	localhost	5004	18046

* Node number is used for brevity for this report only. Nodes don't get labeled with a a number in the network.

Step 1: Assign

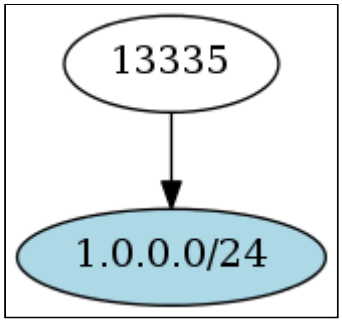
The chain now only has one block, that is the Genesis block. Requesting the chain from a node in the network, e.g., node 0, gives us the following:



The prefix 1.0.0.0/24 is only assigned to one AS (that's node 0), as we can see from the Genesis Assign transaction's output:

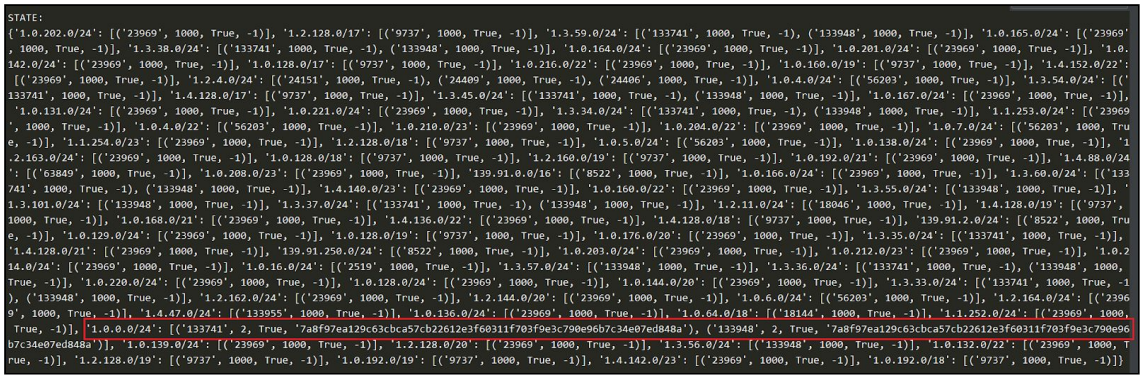
```
chain:
  0:
    hash:          "96dc3b1ed72bb34968fb9dcea9342878c323bb0353ee7b71fdac70d971bdac09"
    index:         0
    miner:         null
    nonce:         0
    previousHash:  -1
    signature:     null
    timestamp:     1549821902.6052773
    transactions:
      0:
        input:     […]
        output:
          0:
            0:     "1.0.0.0/24"
            1:     "13335"
          1:       […]
          2:       […]
          3:       […]
          4:       […]
          5:       […]
          6:       […]
          7:       […]
          8:       […]
          9:       […]
          10:      […]
          11:      […]
          12:      […]
          13:      […]
          14:      […]
```


The AS-level graph for this prefix currently is:

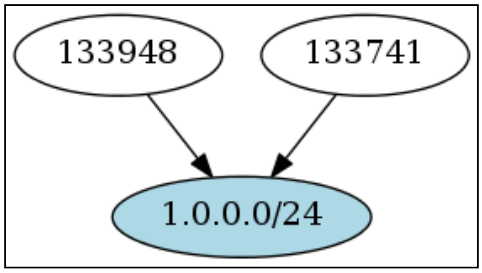


So now we want to transfer the prefix from node 0 to nodes 1 and 2, with ASNs 133741 and 133948, for a period of 2 months and give them the ability to transfer the prefix to other ASes. AS 13335 (i.e., node 0) makes a new Assign transaction and broadcasts it to the rest of the network. The transaction is considered valid and then a new block that includes it is successfully mined. The new block gets added to the chain.

The IP allocation state gets updated after every valid Assign transaction. Now the owners of the prefix are the ASes 133741 and 133948 for a period of 2 months and the original owner has lost any claim over the prefix.



The AS-level graph has been updated as well:

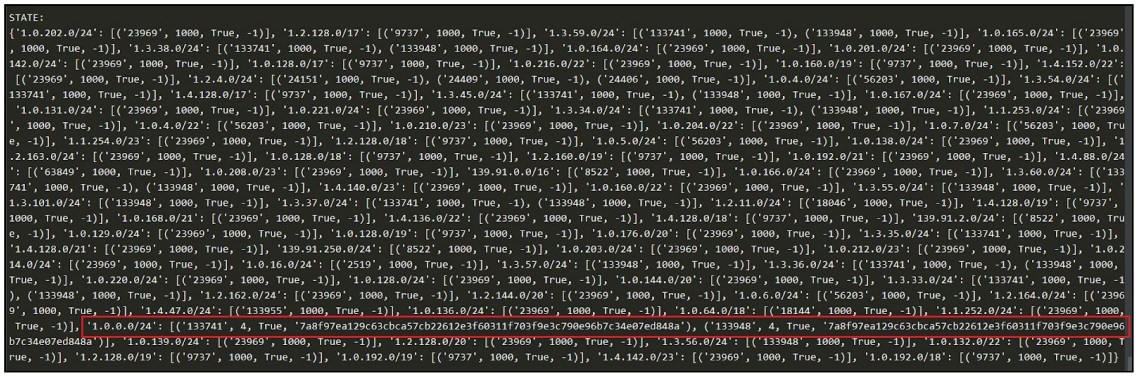


Step 2: Update

We want to update the lease period of the two ASes that got the prefix from the Assign transaction, from 2 to 4 months. By providing the previous valid Assign transaction id found in the chain "7a8f97ea129c63cbca57cb22612e3f60311f703f9e3c790e96b7c34e07ed848a", node 0 makes a new valid Update transaction. After that, a new block is mined and added to the chain that contains this transaction as we can see by requesting the chain:


```
chain:
  0:                    {…}
  1:
    hash:               "0000904df12d72c929ce7a4cd28d5e50c4131b76a577b5d864a4dd0b3281b2c4"
    index:              1
    miner:              "13335"
    nonce:              399
    previousHash:       "96dc3b1ed72bb34968fb9dcea9342878c323bb0353ee7b71fdac70d971bdac09"
    signature:          […]
    timestamp:          1549822912.1925194
    transactions:       […]
  2:
    hash:               "0000b21e56e2238101d31950881cde6f5109c5af705e5f53131cc148fd8f8289"
    index:              2
    miner:              "13335"
    nonce:              17792
    previousHash:       "0000904df12d72c929ce7a4cd28d5e50c4131b76a577b5d864a4dd0b3281b2c4"
    signature:          […]
    timestamp:          1549825260.0591795
    transactions:
      0:
        signature:      […]
        trans:
          input:        […]
          output:
            0:
              0:        "1.0.0.0/24"
              1:        "133741"
              2:        4
              3:        true
            1:
              0:        "1.0.0.0/24"
              1:        "133948"
              2:        4
              3:        true
          timestamp:    1549825257.679693
          txid:         "53c4c2a2d4ab3b3ff61544d2c23ca036b448b649cb327fd2e82b1afddec5a058"
          type:         "Update"
length:                 3
```


The IP allocation state has also been updated after this transaction. As we can see, the lease duration was updated from 2 to 4 months.

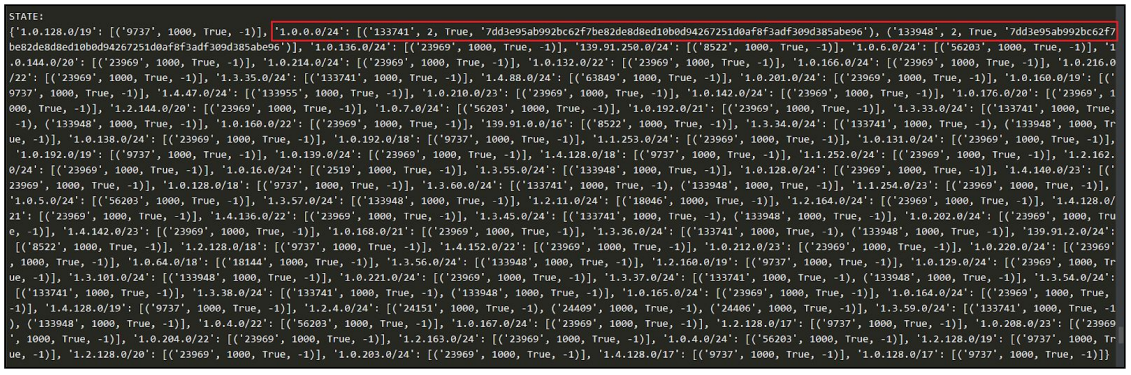


Step 3: Revoke

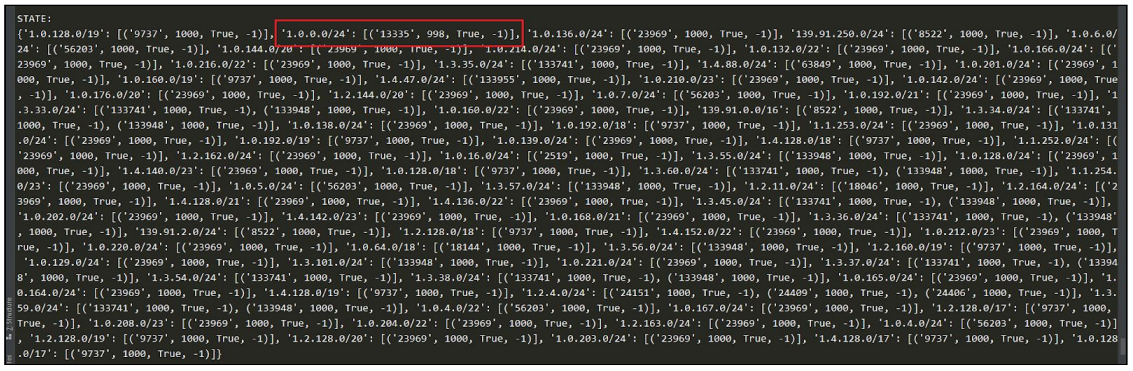
In order to showcase this example, we had to modify a small part in the code so that the we wouldn't have to wait the amount of time needed for the lease to expire. This is a new test case and does not continue after the last test, so this is why the chain is not the same as before. For brevity, we won't discuss the Genesis Assign and the Assign transactions but only the Revoke transaction. The Assign transaction is the same as previously with node 0 assigning the prefix 1.0.0.0/24 to node 1 and 2.
After node 0 transfers the prefix 1.0.0.0/24 to node 1 and node 2 for 2 months, and 2 months go by, the time of the lease expires and so node 0 makes a Revoke transaction. The original owner gets the prefix back and its own leasing period is being updated accordingly.

The IP allocation state before and after the Revoke transaction. As we can see, the previous owners node 1 and node 2 now get replaced by the original owner of the prefix, node 0, and the lease of the original owner is updated.

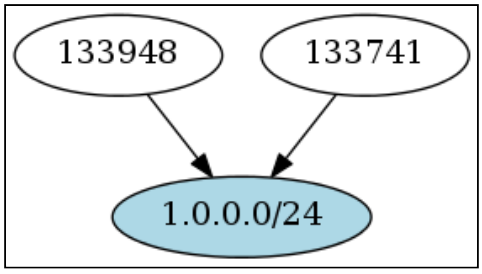
Before the Revoke:



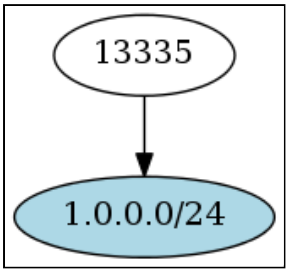
After the Revoke:



The AS-level graph for this prefix before and after the Revoke transaction is the following:



Before



After

BGP Update: Announce & Withdraw transactions

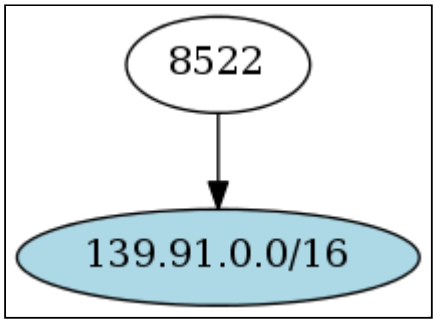
For this test, we make a series of BGP Announce and BGP Withdraw transactions. We replay real BGP updates as BGP Update - Announce transactions, from the same data gathered from BGPStream [5] as mentioned above, and then the node with ASN 8522 makes a BGP Withdraw transaction to withdraw the prefix 139.91.0.0/16 from the ASes it announced the prefix to. We do this in an example workflow of 2 steps (“announce prefix”, “withdraw prefix”).

Step 1: BGP Announce

Replaying the bgp file mentioned above results in the creation of 134 BGP Announce transactions. These transactions are collected in a new block after being evaluated for their validity and the block gets mined and added to the chain.

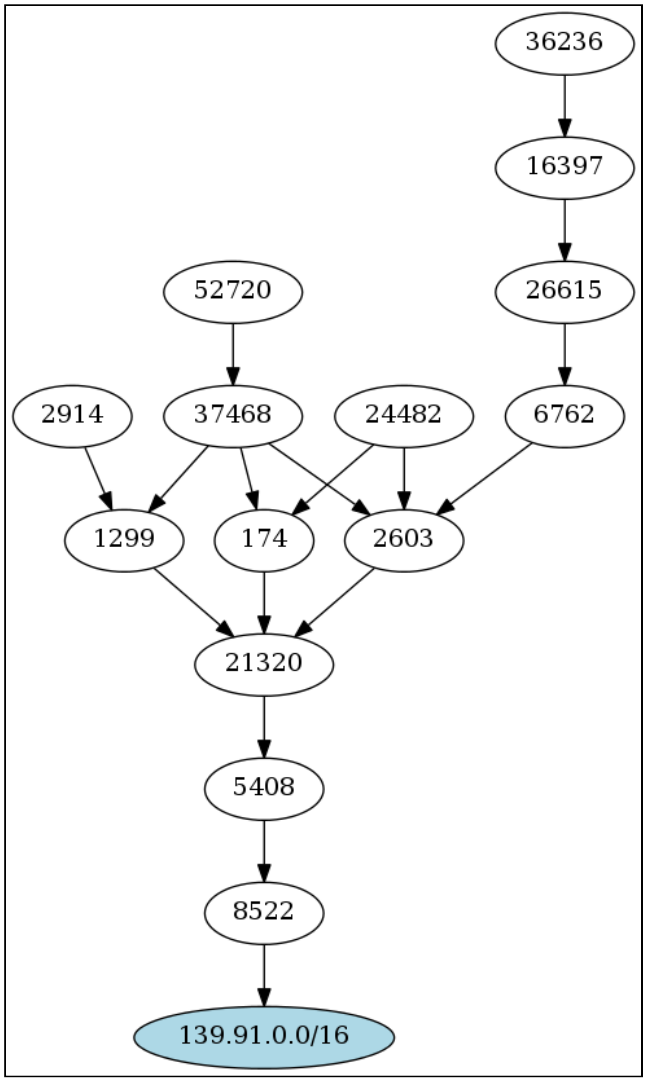

```
chain:
  0:                    {…}
  1:
    hash:               "000017b1816766488eec6a067b7b828bf2e96acc652186dc46546f8a8559b620"
    index:              1
    miner:              "13335"
    nonce:              57306
    previousHash:       "d7bd5d3edae89be40722448c7d0b2fe20e59ce6e444ba6fe20f79dfa66aae3a3"
    signature:          […]
    timestamp:          1550565805.0944872
    transactions:
      0:                {…}
      1:                {…}
      2:                {…}
      3:                {…}
      4:
        signature:      […]
        trans:
          input:
            0:          "139.91.0.0/16"
            1:          "8522"
            2:
              0:        "0"
            3:
              0:        "5408"
              4:        "1532509532"
          output:
            0:
              0:        "139.91.0.0/16"
              1:        "0"
              2:        "8522"
              3:        "5408"
            timestamp:  1550565738.9346023
          txid:         "4eba35790b3bcc280e6948e8019cd06ad20b358722f9b17c1b5ede154265c55e"
          type:         "BGP Announce"
      5:                {…}
      6:                {…}
```


BGP Announce transactions use the AS-level graph state to check for validity; this state gets updated after every new BGP Announce transaction. Before the BGP Announce, the graph of the prefix 139.91.0.0/16 looked like this:



The prefix was mapped to AS with number 8522 as we found out from our input dataset from CAIDA and so the graph was created at the "genesis" bootstrap period.

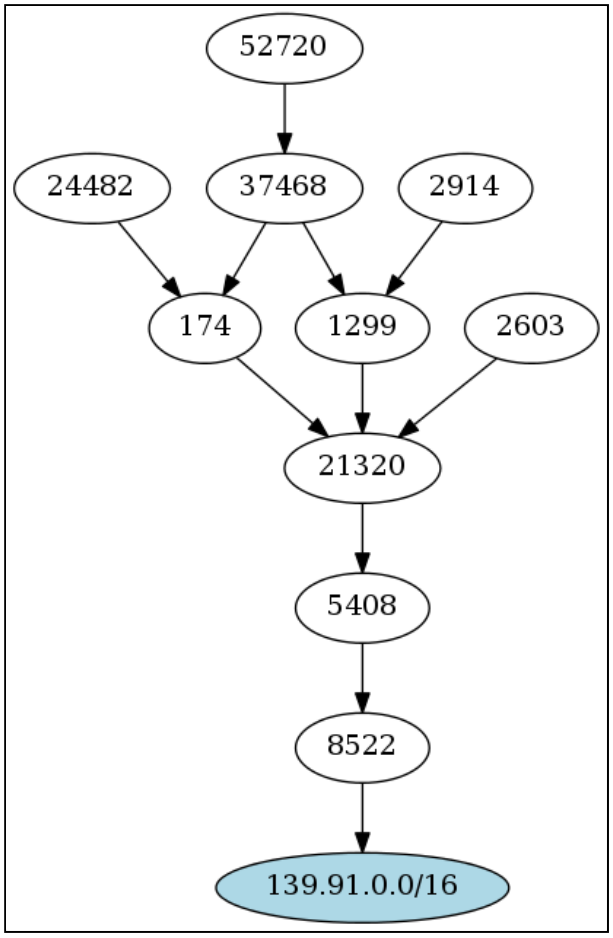
The graph after the series of Announce transactions looks like this:



Step 2: BGP Withdraw

Lets have AS 2603 withdraw the prefix first and then AS 8522. AS 2603 makes a new BGP Withdraw transaction. The transaction is valid and so it will appear in the next block in the chain as we see below:

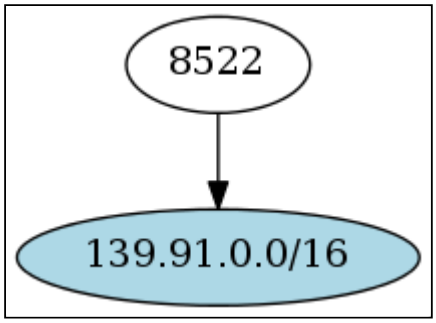
After the BGP Withdraw transaction, the graph of the prefix looks like this:



After AS 8522 makes its own BGP Withdraw transaction, it gets included in the chain in a new block:

```
chain:
  0:                    {…}
  1:                    {…}
  2:
    hash:               "0000aabc2d6be611acaa9a055b8f0292298058e2f9b01a409dadd8fd06b2b52a"
    index:              2
    miner:              "13335"
    nonce:              29848
    previousHash:       "000017b1816766488eec6a06…52186dc46546f8a8559b620"
    signature:          […]
    timestamp:          1550566311.138386
    transactions:       […]
  3:
    hash:               "0000fc6e48d969d8cc3ecf11a555f6e086723791ce4dbe6a3ced89d72a133f42"
    index:              3
    miner:              "13335"
    nonce:              117644
    previousHash:       "0000aabc2d6be611acaa9a055b8f0292298058e2f9b01a409dadd8fd06b2b52a"
    signature:          […]
    timestamp:          1550566772.706841
    transactions:
      0:
        signature:      […]
        trans:
          input:
            0:          "139.91.0.0/16"
            1:          "8522"
            2:          null
          output:
            0:
              0:        "139.91.0.0/16"
              1:        "8522"
          timestamp:    1550566761.0653853
          txid:         "e014cac0f94f973c8ea6d9a55aaee6251506bb20d1bb1bce1a164e989e335a11"
          type:         "BGP Withdraw"
length:                 4
```


And finally, after AS 8522 makes a BGP Withdraw transaction, the graph returns to its original state as it was before the previous announcements.



4.3. Speed

The setup of our experiments in this section, is the following:

- Nodes receive the BGP announcements (out of the raw BGP update data) and replay them as BGP Update - Announce transactions,
- 1 node that mines these transactions into blocks (up to 7 transactions per block, 35 blocks in total).

The measurement target, is to calculate the difference between the time where a block (containing one or more transactions) is actually added to the chain and the time when the transaction was first announced/broadcasted in the BC network. This time difference should be independent from the inter-transaction interval; however, the interval should be large enough to allow for mining to complete. Therefore, for each transaction, we calculate the time difference between the time the block in which the transaction is found at was mined and added to the chain, and the time of creation of the transaction itself. This is indicative of the "speed" of the process, i.e., whether the chain formation can keep up with online management of transactions. This is important for Internet blockchains to avoid the buffering of enormous backlogs (e.g., BGP updates), e.g., during periods of bursty control-plane traffic.

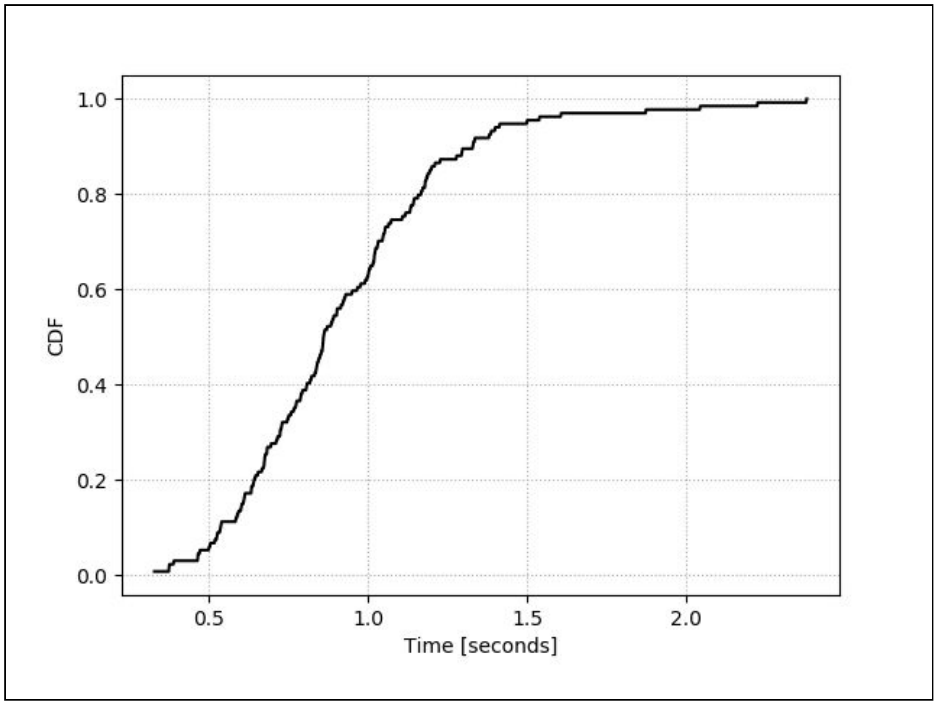
In practice, we observe the following mining times for different Proof-of-Work (PoW) scenarios:

- PoW with 4 zeros: 0.0044 - 1.40 seconds
- PoW with 5 zeros: 0.116 - 12.83 seconds
- PoW with 6 zeros: 0.30 - 223.83 seconds

Replaying the bgp file with the data we gathered from BGPStream as BGP Announce transactions 3 times, gives us the following results. These CDF plots show us how much time passed until one transaction ended up in one block. There is a total of 134 BGP Announce transactions replayed in each of the 3 cases below.

Case 1:

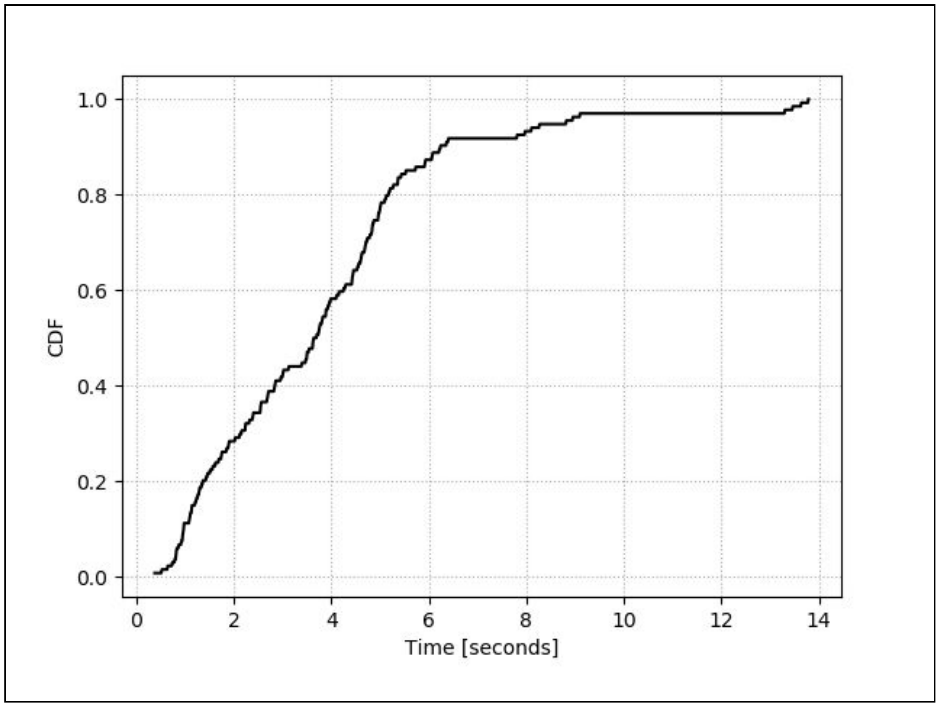
- Proof of Work difficulty: **4** leading zeros in the hash of the block
- Average time of mining of the block: **0.233** seconds
- Median of the CDF: **0.86**



We see that 90% of the transactions need less than 1.5 seconds to be mined, while the rest can reach up to 2.5-3 seconds maximum.

Case 2:

- Proof of Work difficulty: **5** leading zeros in the hash of the block
- Average time of mining of the block: **3.05** seconds
- Median of the CDF: **3.63**

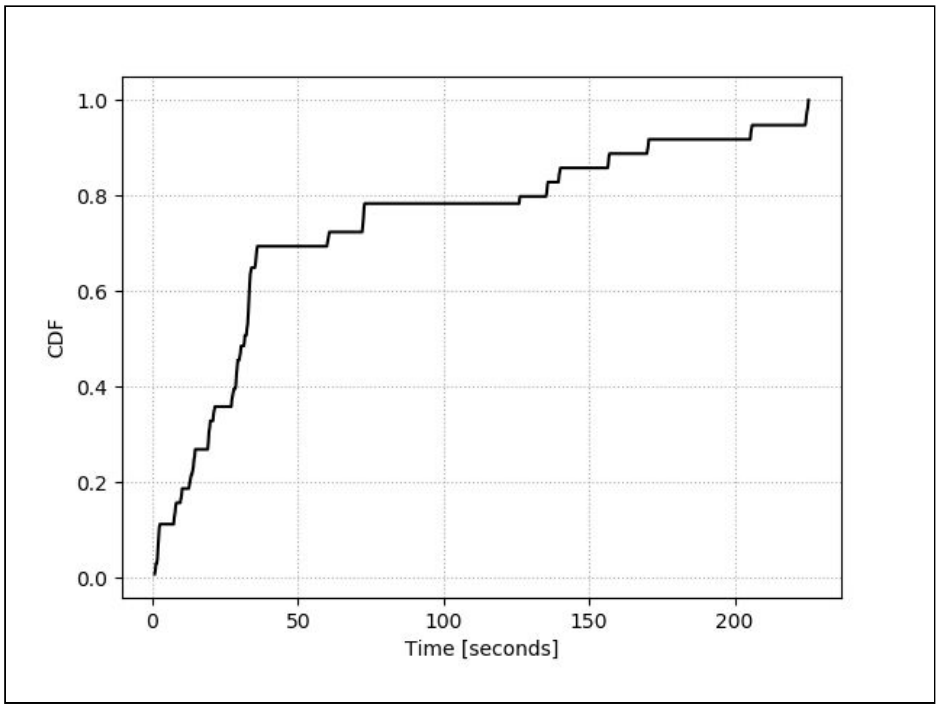


After raising the difficulty up to 5 leading zeros we see that 90% of the transactions need less than 8 seconds to be mined while the rest can reach up to 14 seconds. That is expected since the proof

of work problem is becoming more difficult to solve thus transactions will end up later in the chain.

Case 3:

- Proof of Work difficulty: **6** leading zeros in the hash of the block
- Average time of mining: **54.68** seconds
- Median of the CDF: **31.64**



In this case, the difficulty of the proof of work has been raised to 6 leading zeros. This results in an average time of 55 seconds for the creation of new blocks which indeed as we see affects the speed at which new transactions end up in the chain. 80% of the transactions now need less than 150 seconds to be mined while the rest can reach up to 225 seconds.

4.4. Scalability-Feasibility

4.4.1. BC Storage requirements

In the following, we calculate the average (RAM/HDD) capacity per day that is added to the Blockchain due to BGP Update and IP prefix allocation transactions; we further estimate the total capacity occupied by the Blockchain state primitives (i.e., IP Allocation dictionary and AS-level graph per prefix). Auxiliary state, not contributing significantly to these numbers, is mentioned for completeness. Note that all the following calculations are based on the BGP report by Geoff Huston [15] and data from CAIDA; however, they are back-of-the-envelope calculations for estimating the scalability properties of the chain in terms of space.

4.4.1.1. Transactions and Blocks - rate of increase

BGP update transactions

Estimates according to [15]: O(100k) updates per day. Each AS involved in an update (announcement or withdrawal) learned by one of its neighbors, propagates this update to the rest of its neighbors. In the BC, we group these updates sent to each neighbor in a single transaction stating: “ASX learned sth from ASY and sent it to ASW, ASZ, …”. Therefore, the transaction (vs update) counts are scaled down by a factor approximated by the neighbor degree of each AS. This value is (according to CAIDA) ~5-6 = Θ(10), meaning that the BC should see O(100k/5)=O(10k) transactions per day related to BGP announcements of withdrawals. Note that we do not account for the BGP exploration process which may result to the Announce transactions being actually more than one orders of magnitude less than then BGP updates (since updates with the same sub-paths are encoded once in a transaction), and consider the worst-case scenario for scalability.
For type **BGP Update - Announce**, we have:

- prefix: 32 bit IPv4 (+ 6 bit netmask) | 128 bit IPv6 (+ 16 bit netmask) = O(10) bytes
- bgp_timestamp: 32 bit = O(1) bytes
- adv_AS: 32 bit = O(1) bytes
- source_ASes: list of ASNs = O(1) x Θ(10) = O(10) bytes
- dest_ASes: list of ASNs = O(1) x Θ(10) = O(10) bytes
- timestamp: O(10) bytes
- type: O(1) bytes
- Sum: O(10) bytes

For type **BGP Update - Withdraw**, we have:

- prefix: 32 bit IPv4 (+ 6 bit netmask) | 128 bit IPv6 (+ 8 bit netmask) = O(10) bytes
- ASN: 32 bit = O(1) bytes
- timestamp: O(10) bytes
- type: O(1) bytes
- Sum: O(10) bytes

*⇒ Therefore, per transaction we need O(10) bytes, and therefore 10*O(10k) = O(100k) bytes in total per day. So the blockchain increases by O(100) Kbyte per day due to BGP Update transactions.*

IP prefix transactions

As an approximation, we estimate a rate of increase 2 orders of magnitude smaller than the BGP updates (since IP prefixes change hands through slow management processes and not automatically). Therefore, O(1k) transactions per day related to IP prefix allocations [16].
For type **IP Allocation - Assign**, we have:

- ASN: 32 bit = O(1) bytes
- prefix: 32 bit IPv4 (+ 6 bit netmask) | 128 bit IPv6 (+ 16 bit netmask) = O(10) bytes
- as_dest: list of ASNs = O(1) x Θ(10) = O(10) bytes
- lease duration (in months): O(1) bytes
- transfer tag: O(1) bytes
- last Assign txid: 256bit = O(10) bytes
- lease duration of source: O(1) bytes
- timestamp: O(10) bytes
- type: O(1) bytes
- Sum: O(10) bytes

For type **IP Allocation - Revoke**, we have:

- ASN: 32 bit = O(1) bytes

- IP Allocation - Assign txid: 256bit = O(10) bytes
- timestamp: O(10) bytes
- type: O(1) bytes
- Sum: O(10) bytes

For type **IP Allocation - Update**, we have:

- ASN: 32 bit = O(1) bytes
- IP Allocation - Assign txid: 256bit = O(10) bytes
- lease duration (in months): O(1) bytes
- timestamp: O(10) bytes
- type: O(1) bytes
- Sum: O(10) bytes

*⇒ Therefore, per transaction we need O(10) bytes, and therefore 10*O(1k) = O(10k) bytes in total per day. So the blockchain increases by O(10) Kbyte per day due to IP Allocation transactions.*

4.4.1.2. State - needed capacity

IP prefix state

We have in total, according to [15]:

- O(100k) IPv4 prefixes → each prefix takes 32 bit (net address) + 6 bit (net mask) = 38 bit = O(1) bytes
- O(10k) IPv6 prefixes → each prefix takes 128 bit (net address) + 16 bit (net mask) = 144 bit = O(10) bytes
- O(10k) ASes originating IPv4 + IPv6 prefixes → each ASN takes 32 bit = O(1) bytes

We maintain per prefix:

- ASN (32 bit; O(1) bytes)
- Lease duration (in months) (32 bit; O(1) bytes)
- transfer tag (1 bit; O(1) bytes)
- txid (256 bit; O(10) bytes)
- Sum: O(10) bytes

⇒ So, in the worst case, we need:
*O(100k*1*10k*10 + 10k*10*10k*10) = O(10^10) = O(10) Gbytes for maintaining the IP Allocation state. This can be stored as a Python dictionary in RAM, and requires (i) either the necessary RAM to be available on the machine, or (ii) technical way to store (part of) the dictionary in HDD.*

Note that this estimation is compliant (O(10) GB for ~800K prefixes) with the empirically measured blockchain size (~1GB for ~150K IP prefixes) found in [17], meaning that while we follow a different approach and architecture, our calculations correctly approximate the order of magnitude needed to store this kind of state.

AS-level graph state

We have in total, according to [15]:

- O(100k) IPv4 prefixes → each prefix takes 32 bit (net address) + 6 bit (net mask) = 38 bit = O(1) bytes

- O(10k) IPv6 prefixes → each prefix takes 128 bit (net address) + 16 bit (net mask) = 144 bit = O(10) bytes
- O(10k) ASes originating IPv4 + IPv6 prefixes → each ASN takes 32 bit = O(1) bytes
- O(100k) AS-level links → each link takes 2*32bit=64bit=O(1) bytes

⇒ So, in the worst case, since we keep one AS-level graph per prefix (and the edges dominate in terms of numbers), we need:
*O(100k*1*100k*1 + 10k*10*100k*1) = O(10^10) = O(10) Gbytes for maintaining the AS-level graph state. This can be stored as a Python dictionary in RAM, and requires (i) either the necessary RAM to be available on the machine, or (ii) technical way to store (part of) the dictionary in HDD.*

4.4.1.3. Other auxiliary state (<< prefix and graph states)

The following states are mentioned for completeness purposes (they do not affect the big O calculations done above).

- **txid_to_block:**
  Dictionary with keys the transaction ids and values the block index. transaction ids are 256 bit : block index 32 bit. Increases at the same rate as the chain; storing a single key and value requires O(10) bytes.
- **pending_transactions:**
  List of some transactions before they get added to the chain. Gets cleared frequently, after every mining; does not affect permanent storage requirements.
  Size varies based on type of transaction.
- **invalid_transactions:**
  A List of 256 bit transaction ids. This does not get cleared for security purposes, but its initial O(10) byte size increases only if some node behaves maliciously (and we can further tune the algorithm to keep only the most recent invalid transactions).
- **my_assignments:**
  A set with the 256 bit txids of all the assign transactions a node has made. Its O(10) byte initial size increases together with the blockchain, depending on the local node assignments.
- **update_sum, assign_sum:**
  Dictionary state. Key 32 bit ASN : value 32 bit int. Cleared after mining; does not affect permanent storage requirements.
- **bgp_txid_announced:**
  Dictionary with keys 256 bit txids and values 1 bit True/False. Used for checking duplicate transactions. Its O(10) byte initial size increases together with the blockchain.
- **as_to_announced_txids:**
  Dictionary with keys 32 bit ASN and values: a list of 256 bit txids. Similar as with “my_assignments”.
- **assigned_prefixes:**
  Set with the prefixes a node has assigned to others. 38 bit IPv4 and 128 bit IPv6. Cleared after mining; does not affect permanent storage requirements.
- **assign_txids:**
  Set with the 256 bit Assign transaction ids of a node. Cleared after mining; does not affect permanent storage requirements.
- **alive_neighbors:**
  Dictionary with keys url of a node (IP:port; O(10) bytes)) and value a float timestamp in

python (O(10) bytes). It increases linearly with the size of the network (in terms of node counts).

4.4.2. Optimizations for Internet scales

4.4.2.1. Duplicate update/transaction checks

There are checks implemented that do not allow for duplicate IP Assign transactions and BGP Announce to be made even before a new block is mined. By using auxiliary states we make sure that there are no duplicates of these transactions made. This helps with maintaining the overall size of the blockchain to the minimum since these types of transactions are usually the ones that carry the most data.

4.4.2.2. Bulk transactions per block

Multiple transactions can be bundled together in the same block. The transactions can be of different types such as an IP Assign followed by a BGP Announce. This should help with keeping the amount of blocks in relatively low numbers so that we don't have to mine new blocks for every single transaction; however we should be careful and mine new blocks relatively fast so that the honest chain can still outpace attackers. Therefore, there is a trade-off between aggregating transactions and generating large numbers of blocks.

5. Conclusions and future work

5.1. Summary of Thesis

In this thesis, we studied the principles of the blockchain technology and how it could be used for managing key resources of the Internet. We studied the basics of BGP updates, how current security mechanisms fail to protect IP resources and why there is a need for a distributed tamper-resistant ledger to help secure these resources in the Internet.

We implemented a viable blockchain-based prototype framework as a solution for the management of IP prefix allocations and BGP updates that eliminates the need for any PKI dependency or a centralized authority. While the use of a blockchain, for our purposes, is to help us secure key resources in the Internet, it has a lot other advantages as well such as transparency and immutability, the most important being disintermediation; it enables a database to be directly shared without the need of a central administrator. We verified the correctness of our approach with several tests and checks.

The main challenge, however, in using a blockchain is its scalability. To investigate this, we estimated the speed at which an IP allocation or BGP update transaction can be mined into a block; in our experiments, and depending on the Proof of Work implementation (e.g., difficulty), this can range from 1.5 to 225 seconds. We understand that this speed is not enough to cope with the load of 1000s of BGP updates per second, if we consider a single BGP Update transaction per block; however, note that a single block may contain 1000s of transactions and the actual rate of BGP Update transactions is much lower than the rate of the raw BGP updates. While more research is needed for assessing this, the preliminary results we have are promising.

Moreover, with some preliminary back-of-the-envelope calculations, we deduced that with moderate resources per AS (O(10) Gbytes storage, increasing by O(100) kbytes per day), we can in theory keep up with the current load in inter-domain routing; even if our calculations are off by one or even two orders of magnitude, the resources are still manageable. However, further evaluation needs to be done in this direction, accounting also for hidden overheads or practical constraints. Research on scaling up blockchains is a very active topic in the community; advances there would also benefit our research on the Internet blockchain.

Most importantly, we transferred the idea of the Internet blockchain from concept to design and implementation. We plan to open-source our code and evaluation scripts and make the prototype available to the research community.

In conclusion, this thesis offered me a good opportunity to learn about BGP and the security issues the protocol faces, the Blockchain technology that cryptocurrency depends on and how it could help revolutionize the world. It also offered me the chance to improve my programming skills by undertaking such a big project and also gave me the motivation to learn a new programming language by myself. Finally, by writing this report I was able to improve my writing skills and learn about the process of writing a technical report.

5.2. Future Work

Experiments on scalability with larger IP Allocation and BGP Update transaction load

For simplicity, in our experiments, we used a smaller sample of the IP Allocation data we acquired from CAIDA [18]; the BGP Update transaction load that we generated in our network was relatively light in order to easily verify that at this stage everything is working properly. In the future, however, we should run experiments with larger IP Allocation data and generate heavier BGP Update transaction loads; we then need to check how our framework can handle this and find other potential bugs or make new improvements. For example, we can also investigate other APIs and protocols that could be used for the communication between the nodes (taking into account that these nodes are AS agents), besides extending the REST API that we currently support.

Proof of Stake (PoS) vs Proof of Work (PoW)

Proof of Stake (PoS) is an algorithm that a blockchain network could adopt to achieve distributed consensus. This algorithm is choosing the creator of the next block via various combinations of random selections such as the wealth or age; this is defined as stake. The PoW algorithm uses mining; a task where a miner is solving computationally intensive problems to create new blocks. The miner who solves the problem first wins. Our prototype uses the PoW algorithm as the consensus algorithm for creating new blocks.

One of the main drawbacks of the proof of work algorithm is that the miners consume a lot of electricity [12, 13]. One Bitcoin transaction requires the same amount of electricity as powering 1.5 American households for one day (data from 2015) [10]. A research work showed that Bitcoin could consume as much electricity as Denmark by 2020 [11]. So that means we have to find an alternative consensus algorithm that is also “greener”. There are criticisms [20] that PoS is not an ideal option for a distributed consensus protocol. PoS introduces many new problems such as the "nothing-at-stake" problem; wherein block generators have nothing to lose by voting for multiple

blockchain histories, thereby preventing consensus from being achieved. And unlike in PoW systems, one node can work on several chains in little cost. Ethereum [22] suggested a PoS protocol named "Slasher" but it was never adopted. Ethereum developers concluded PoS is "non-trivial" [21], some said even impossible [24] and they opted for a PoW algorithm. It is planned, however, to transition from a pure PoW algorithm to a PoS/PoW hybrid named "Casper" [24].

In conclusion, we see that both PoW and PoS have advantages and both introduce problems that need to be addressed. However, there is a lot of ongoing research in this field and it will be interesting to see how these problems will be finally solved, how we could transition to a more green algorithm that is secure and does not require a lot of resources and finally how we could incorporate this new algorithm into our own blockchain!

Genesis Blocks from Official Registrars

For the purposes of this thesis (Proof of Concept) we used input for the genesis blocks, i.e., the initial IP prefix allocations from CAIDA, and specifically from its prefix-to-ASN mapping dataset [18]. However, the latter are empirically collected BGP data (from RouteViews route collectors), and while they are rich and useful for our purposes, they could be tainted (e.g., by corrupt hijack data) and do not constitute a “ground truth”. In future work, we would like to collect prefix-to-ASN mappings from official registrars, like RIPE NCC, so that we have the legally binding mappings that are publicly available. Another source of such authoritative data would be RPKI registries.

Selective Withdrawals

Right now, after an AS withdraws a prefix, we remove from the AS-level graph the edges between the withdrawing node and its predecessors (those are the nodes to which the withdrawing node had previously advertised the prefix to) and all the nodes that cannot reach the prefix anymore. A feature that we could implement in the future would be to allow an AS to withdraw the prefix selectively, thus deleting only some connections to its neighbors.

Selective Updates, Revokes

IP allocation Update and Revoke transactions, currently, update or revoke the prefix from all the ASes it was previously assigned to. One thing it could be done is to change these 2 types of transactions so that they could support selective updates or revokes.

Incremental Deployment and Bootstrapping

While the system we have designed and implemented is demonstrated as a Proof of Concept, one of the biggest challenges of adopting such approaches in the real Internet is incremental deployment, e.g., how to bootstrap the system in the wild. We believe that the IP Allocation transactions could be bootstrapped using official Internet registrars to generate the initial Genesis blocks; content or hosting providers and DDoS protection organizations could then join and add to the chain to protect their own prefix ownership claims in a structure that stands in parallel to RPKI; ISPs and other stub networks would then start joining. Receiving BGP Update transactions is a bit more complex; validating BGP paths requires deployment along contiguous “chains” of ASes that run the corresponding agents. We believe that this could be bootstrapped as a service offered between edge networks and their direct providers, validating the corresponding first-hop peering

links, as well within IXPs to validate multilateral peering sessions. With such an initial deployment, increments could be done by following the current Internet relationships (customer to provider to peer to peer to customer, etc.), and forming contiguous chains of ASes that are eventually translated in unbroken AS-level paths and associated BGP Update transactions. For separating concerns and due to the different features of IP Allocation and BGP Update transactions (slow vs fast changing, respectively), we could consider having two separate chains; one per transaction type. The BGP Update chain could then be reset in periodic intervals of time (e.g., every N months) in agreement with the involved nodes, in order to preserve scalability attributes and not keep stale information. Finally, the rich infrastructure of BGP route collectors that are available in the Internet could help provide BGP update and AS-path data to compare the chained transactions against, which is useful for evaluating the system's operation during its initial deployment stages, where a critical mass of deployment on the AS-level Internet has not been yet reached. The system could also be used to complement current BGP hijack detection systems [19].

Make Routing depend on BC validation

While our system works in parallel to inter-domain routing, validating IP Allocation and BGP Update transactions online and in a passive fashion, we could think of critical extensions that make actual routing dependent on this validation. For example, the system could instruct a BGP router to reject an incoming route advertisement, if the route has failed the blockchain transaction validity checks. However, this would require a critical mass of deployment first (see previous point), and much more work would be needed to research scalability limitations (especially regarding time, e.g., fast convergence) as well as the implications of rejecting routes in case e.g., the chain has been corrupted or compromised by an attacker. Therefore, this remains an interesting research direction to explore; blockchain is one way to solve the path validation problem but may not be the best one.

Recording DNS Transactions in the Blockchain

We could follow a similar concept and include DNS name to IP address mappings in our blockchain. We could make a new type of transaction the DNS transactions that hold DNS names to IP address mapping data.That way domains could be protected from being spoofed. Private domains do not need to be included in the chain as they are not publicly accessible. This was proposed in the Internet Blockchain paper [3] in section 2.5.

6. References

[1] Satoshi Nakamoto, "Bitcoin: A Peer-to-Peer Electronic Cash System", https://bitcoin.org/bitcoin.pdf

[2] Stuart Haber, W. Scott Stornetta, "How to Time-Stamp a Digital Document", https://www.anf.es/pdf/Haber_Stornetta.pdf

[3] Adiseshu Hari, T.V. Lakshman, "The Internet Blockchain: A Distributed, Tamper-Resistant Transaction Framework for the Internet http://web.kaust.edu.sa/Faculty/MarcoCanini/classes/CS390G/S17/papers/InternetBlockchain.pdf

[4] Cloudflare, "What Is BGP? | BGP Routing Explained",

https://www.cloudflare.com/learning/security/glossary/what-is-bgp/

[5] BGPStream, "An open-source software framework for live and historical BGP data analysis, supporting scientific research, operational monitoring, and post-event analysis.", https://bgpstream.caida.org/docs

[6] Resource Public Key Infrastructure (RPKI), https://www.apnic.net/get-ip/faqs/rpki/

[7] BGP security: the BGPsec protocol, https://www.noction.com/blog/bgpsec_protocol

[8] A Survey among Network Operators on BGP Prefix Hijacking, Pavlos Sermpezis, Vasileios Kotronis, Alberto Dainotti, Xenofontas Dimitropoulos https://www.ics.forth.gr/_publications/hijacking_survey_CCR2018.pdf

[9] YouTube Hijacking: A RIPE NCC RIS case study, ripe

[10] Article for Motherboard, by Christopher Malmo Bitcoin Is Unsustainable

[11] Article by Sebastiaan Deetman, Bitcoin Could Consume as Much Electricity as Denmark by 2020

[12] Bitcoin Energy Consumption Index, https://digiconomist.net/bitcoin-energy-consumption

[13] Bitcoins Energy Consumption An Unsustainable Protocol That Must Evolve? http://blockgeeks.com/bitcoins-energy-consumption/

[14] Article for The Register, by Richard Chirgwin Japan Internet outage

[15] BGP in 2017, by Geoff Huston https://labs.apnic.net/?p=1102

[16] Jordi Paillissé, Albert Cabellos, Vina Ermagan, Alberto Rodríguez, Fabio Maino, "Could Blockchain Help in Inter-domain Security?", https://ripe76.ripe.net/presentations/121-Blockchain-opensource-ripe76.pdf

[17] Jordi Paillisse, Miquel Ferriol, Eric Garcia, Hamid Latif, Carlos Piris, Albert Lopez, Brenden Kuerbis, Alberto Rodriguez-Natal, Vina Ermagan, Fabio Maino and Albert Cabellos, "IPchain: Securing IP Prefix Allocation and Delegation with Blockchain", https://arxiv.org/abs/1805.04439

[18] Routeviews Prefix to AS mappings Dataset (pfx2as) for IPv4 and IPv6 https://www.caida.org/data/routing/routeviews-prefix2as.xml → Date: 28/03/2018 http://data.caida.org/datasets/routing/routeviews-prefix2as/2018/03/

[19] P. Sermpezis, V. Kotronis, P. Gigis, X. Dimitropoulos, D. Cicalese, A. King, and A.Dainotti, “ARTEMIS: Neutralizing BGP Hijacking Within a Minute,” IEEE/ACM Transactions on Networking, vol. 26, iss. 6, 2018.

https://www.caida.org/publications/papers/2018/artemis/artemis.pdf

[20] Hugo Nguyen, "Proof-of-Stake & the Wrong Engineering Mindset", medium.com/@hugonguyen/proof-of-stake-the-wrong-engineering-mindset-15e641ab65a2

[21] Vitalik Buterin, "Slasher Ghost, and Other Developments in Proof of Stake" https://blog.ethereum.org/2014/10/03/slasher-ghost-developments-proof-stake/

[22] Ethereum, A Blockchain App Platform, https://www.ethereum.org/

[23] Andrew Poelstra, A Treatise on Altcoins, [section 6.4 Proof of Stake] https://download.wpsoftware.net/bitcoin/alts.pdf

[24] "The Ethereum Killer Is Ethereum 2.0: Vitalik Buterin's Roadmap" article by Giulio Prisco https://bitcoinmagazine.com/articles/ethereum-killer-ethereum-20-vitalik-buterins-roadmap/

[25] Yossi Gilad, Avichai Cohen, Amir Herzberg, Michael Schapira, Haya Shulman, "Are We There Yet? On RPKI's Deployment and Security", https://eprint.iacr.org/2016/1010.pdf

[26] Jordi Paillisse, Miquel Ferriol, Eric Garcia, Hamid Latif, Carlos Piris, Albert Lopez, Brenden Kuerbis, Alberto Rodriguez-Natal, Vina Ermagan, Fabio Maino and Albert Cabellos, "IPchain: Securing IP Prefix Allocation and Delegation with Blockchain" https://arxiv.org/abs/1805.04439

[27] Qianqian Xing, Baosheng Wang, Xiaofeng Wang, "BGPcoin: Blockchain-Based Internet Number Resource Authority and BGP Security Solution" https://www.mdpi.com/2073-8994/10/9/408