Imperial College London

MEng Individual Project Final Report

Department of Computing

Imperial College of Science, Technology and Medicine

Sher₿lock Holmes: Digital Blockchain Forensics

Author:
Callum Eden

Supervisor:
Prof. William J Knottenbelt

Second Marker:
Prof. Yves-Alexandre de Montjoye

June 16, 2019

Submitted in partial fulfillment of the requirements for the Masters Of Engineering of Imperial College London

Abstract

The uptake of cryptocurrencies has soared in recent years; proliferating in popularity and mainstream adoption. At the dismay of law enforcement, adopters of this new technology include criminals wishing to evade the regulatory oversight of traditional payment mechanisms to participate in illegal trade (including, but not limited to, drugs, hacks, ransomware and even murder-for-hire [1]). The largest cryptocurrency of all, Bitcoin, is estimated to be involved in $76 billion of illegal activity each year [1]. Those using Bitcoin for illegal means aren't trivially identifiable; Bitcoin identities are pseudo-anonymous, such that they are not tied to any real-world entity, but all transactions they are involved in are publicly visible and entirely transparent.

In this project, we develop a tool to assist with conducting digital forensic investigations across Bitcoin. Although there do exist some commercial tools which have the same goal, they are proprietary and not available to the wider community. To develop our tool, we leverage the transparency of the Bitcoin Blockchain to build a graph database which stores, and represents the relationships between, Bitcoin's 420,000,000+ transactions. The graph database is the foundation to building *Radar*, a web-based digital forensic investigation tool. *Radar* presents historical Bitcoin transactions through interactive, graph-based visualisations. With the support of intuitive graph-navigation and filtering controls, *Radar* can be used to navigate historical Bitcoin activity efficiently. We further Bitcoin associate addresses with the users which control them by using existing public datasets and by using clustering heuristics based on Bitcoin idiom-of-use. Working with investigators at the Metropolitan Police and Coinbase, we show *Radar's* effectiveness in identifying suspicious transactions and tracking the flow of funds.

Acknowledgements

I am incredibly grateful to my supervisor William Knottenbelt for his guidance and enthusiasm in defining the potential of this project.

A big thank you to Alexei Zamyatin for his assistance, on several occasions, in facilitating my access to the hardware resources required for this project.

Finally, I would like to thank Mat Stanley from the Metropolitan Police and Iggy Azad from Coinbase for their critique and insight that helped to evaluate this project and scope out what the future for such an investigation tool could look like.

Contents

1 Introduction **1**
- 1.1 Bitcoin . . . 2
- 1.2 Contributions Outline . . . 2

2 Background **4**
- 2.1 Bitcoin . . . 4
  - 2.1.1 Addresses, Keys & Hashing . . . 4
  - 2.1.2 Bitcoin Address Types . . . 4
  - 2.1.3 Vanity Addresses . . . 5
  - 2.1.4 Blockchain . . . 5
  - 2.1.5 Mining . . . 5
  - 2.1.6 Coinbase . . . 6
  - 2.1.7 Proof of work . . . 6
  - 2.1.8 Transactions . . . 7
  - 2.1.9 Nodes . . . 8
  - 2.1.10 Immutable History . . . 8
  - 2.1.11 Mining Pools . . . 8
  - 2.1.12 Forks . . . 9
  - 2.1.13 Bitcoind . . . 10
- 2.2 Anonymity . . . 10
  - 2.2.1 Mixing Services . . . 10
  - 2.2.2 Risks of Using Transaction Anonymisers . . . 10
  - 2.2.3 Peeling Chain . . . 11
  - 2.2.4 Taint Analysis . . . 11
  - 2.2.5 TOR . . . 12
- 2.3 Bitcoin Address Clustering (on-chain) . . . 12
  - 2.3.1 Multi-Input Transactions . . . 12
  - 2.3.2 Change Addresses . . . 12
  - 2.3.3 Consumer Wallet Heuristic . . . 13
  - 2.3.4 Optimal Change Heuristic . . . 13
  - 2.3.5 Behaviour Based Analysis . . . 14
- 2.4 Bitcoin address clustering (off-chain) . . . 14
  - 2.4.1 Tag Collection . . . 14
  - 2.4.2 Entity Clustering . . . 14
- 2.5 Popular Services . . . 15
  - 2.5.1 Satoshi Dice . . . 15
  - 2.5.2 Exchanges . . . 15
- 2.6 Illegal Activity . . . 15

2.6.1 Silk Road . . . 15
2.6.2 Money Laundering . . . 16
2.6.3 Bitcoin ATM's . . . 16
2.6.4 Significant Thefts . . . 17
2.7 Existing Forensic Tools . . . 17
2.7.1 Blockchain Explorer . . . 17
2.7.2 Chainanalysis . . . 18
2.7.3 Wallet Explorer . . . 18
2.7.4 Blockpath . . . 19
2.7.5 Other Solutions . . . 20
2.8 Importing Blockchain Data . . . 20
2.8.1 Bitcoin to Neo4J Tool: Open Source Project . . . 20
2.8.2 Max Baylis: Imperial MSc Project 2018 . . . 21
2.8.3 TokenAnalyst: Medium Blog . . . 21
2.8.4 Blockchain2graph: Open Source Project . . . 21
2.8.5 Analysis of Previous Work . . . 21
2.9 Know Your Customer . . . 21
2.10 Privacy Enhanced Cryptocurrencies . . . 22
2.10.1 ZCash . . . 22
2.10.2 MimbleWimble (Protocol) . . . 22
2.10.3 Dash . . . 22
2.10.4 Monero . . . 22
2.11 Technology . . . 23
2.11.1 Spring WebFlux . . . 23

3 Blockchain Download: *Astrolabe* **24**
3.1 Hardware . . . 24
3.2 Retrieving Historical Bitcoin Transactions . . . 24
3.3 Challenges & Solutions: . . . 25
3.3.1 Efficiency . . . 25
3.3.2 Job Failure Mitigation . . . 26
3.3.3 Writing Concurrently from Several Threads . . . 26
3.3.4 Duplicate Addresses . . . 26
3.4 Result . . . 27

4 Fetching Historical Price Data: *Compass* **28**
4.1 Source of Price Data . . . 28
4.2 Storing the Price Data . . . 28
4.3 Matching Price Data to Bitcoin Data . . . 28
4.4 Using the Price Data . . . 29

5 Entity Tagging: *Quadrant* **30**
5.1 Retrieving Wallet Data . . . 30
5.1.1 Building the Scraper . . . 30
5.2 Results . . . 31
5.3 Performing the Address Matching . . . 32

6 Database Population **33**
6.1 Why Neo4J? . . . 33
6.1.1 Other DB Solutions . . . 33
6.1.2 Bulk Import Tool . . . 34
6.2 Database Design . . . 34
6.2.1 Data Nodes . . . 34
6.2.2 Relationships . . . 34
6.3 Invoking the Import Job . . . 36
6.4 Challenges & Solutions . . . 36
6.4.1 Memory Issues . . . 36
6.4.2 Query Latency Issues . . . 36
6.4.3 Creating indexes . . . 37
6.5 Import Result . . . 37

7 Backend API: *Loran* **38**
7.1 Technology Choices . . . 38
7.1.1 Alternative Technologies . . . 38
7.2 API Design . . . 38
7.2.1 Responses . . . 40
7.3 Implementation . . . 41
7.3.1 Overall Design . . . 41
7.3.2 Node Entities . . . 43
7.3.3 Serialising Node Entities . . . 43
7.3.4 Implementing Repositories . . . 44
7.3.5 Implementing Path Finding . . . 44

8 Clustering: *Balestilha* **45**
8.1 Algorithm . . . 45
8.2 Java & Spring Data Approach . . . 45
8.2.1 Challenges . . . 47
8.3 Cypher Query . . . 48
8.3.1 Challenges . . . 49
8.4 Clustering on demand . . . 49
8.4.1 Challenges . . . 51
8.5 Clustering using raw CSV data . . . 52

9 Investigation Tool: *Radar* **53**
9.1 Technology Choices . . . 53
9.1.1 Angular 6 & TypeScript . . . 53
9.1.2 D3 . . . 54
9.2 Implementation . . . 54
9.2.1 Routes . . . 54
9.2.2 Architecture . . . 55
9.3 Features . . . 58
9.3.1 Search by Address . . . 58
9.3.2 Search by Entity Name . . . 59
9.3.3 Node Information on Hover . . . 61
9.3.4 Link Data . . . 61
9.3.5 Traverse the Graph . . . 62

9.3.6 Link Dependant Colour and Size of Nodes 63
9.3.7 Selecting Fiat Currencies 63
9.3.8 Filter by Date and Time 64
9.3.9 Filter by Value in Several Currencies 65
9.3.10 Limiting Nodes 65
9.3.11 Enable *Multi-Input* Clustering View 66
9.3.12 User Input Validation & Feedback 68
9.3.13 Add Custom Nodes 68
9.3.14 Link Custom Nodes to Other Nodes 69
9.3.15 Path Finding 70
9.3.16 Persisting Data 71

10 Overall Deployment 72
10.1 Developing on Satoshi 74

11 Evaluation 75
11.1 Meeting Investigators from Industry 75
11.1.1 Successes and Weaknesses 75
11.1.2 Desirable Features 76
11.1.3 Additional Data Sources 77
11.2 Performance 77
11.2.1 Individual Cypher Query Profiling 77
11.2.2 Performance Under Load 79
11.2.3 Blockchain Import 82
11.3 Performing a Historical Investigation 83
11.3.1 Areas identified for improvement 88
11.4 Path Finding Correctness 88
11.5 Clustering Correctness 89
11.6 Missed Objectives 90
11.7 Comparisons with existing tools 90
11.7.1 Wallet Explorer 90
11.7.2 Blockchain Explorer 92
11.7.3 Blockpath 92
11.8 Risks 94
11.9 Summary 94
11.9.1 Weaknesses 94
11.9.2 Strengths 94

12 Conclusion 96
12.1 Reflection 96
12.1.1 Future of Crypto-Currency Law Enforcement 96
12.1.2 Working with Data at Scale 97
12.2 Future Work 97
12.2.1 Infrastructure for Keeping Database up to Date 97
12.2.2 Path Finding User Experience 98
12.2.3 Incorporate Information from more Sources 98
12.2.4 Saving Investigations 99
12.2.5 Exporting Investigation Data 99
12.2.6 Change Address Clustering Heuristic 99

12.2.7 More Clustering Heuristics . . . 100
12.2.8 Set up Watches for Nodes . . . 100
12.2.9 UI Improvements . . . 100
12.2.10 Several Crypto-Currencies . . . 101

Appendices **105**

A User Guide : *Radar* **106**
A.1 Search . . . 106
A.2 Investigate . . . 107

B *Radar* Views **108**

C Locust Evaluation Results **111**

D Terminology **117**

Chapter 1

Introduction

Cryptocurrencies provide us with a mechanism to send a unit of cryptocurrency to anyone, directly, anywhere. Cryptocurrencies are entirely virtual; there are no physical coins or even digital coins per se. There are no banks involved, no mediators and no central reserve controlling coin supply; what cryptocurrencies often have, however, are cryptography principles such as elliptic curve cryptography and one-way hash functions that help secure ownership of funds. Cryptocurrencies can be classified by their ability to establish ownership, protecting against double spending, ensuring anonymity/privacy and minting new currency [2].

There are several leading cryptocurrencies which, together, dominate the cryptocurrency market capitalisation [see figure 1.1]. The biggest player of all leading cryptocurrencies is Bitcoin.

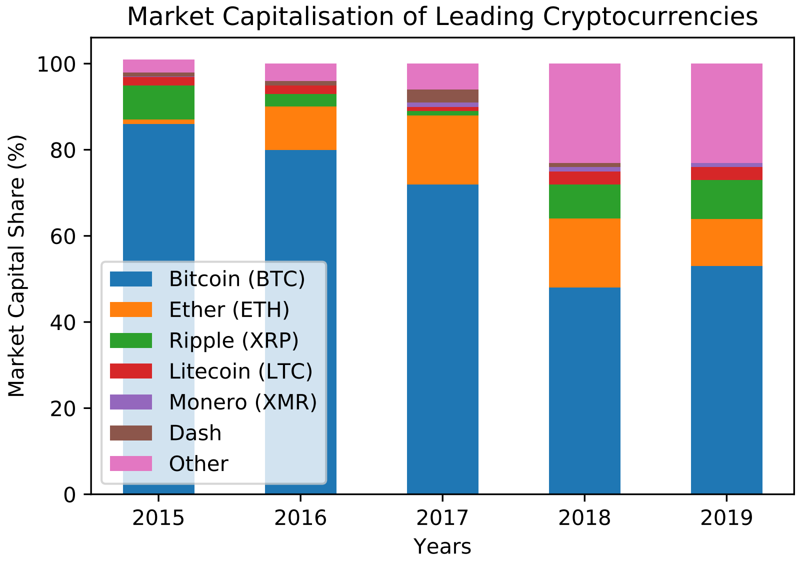

Figure 1.1: Market Capitalisation of leading Cryptocurrencies from 2015-2019 [3]

1.1 Bitcoin

Bitcoin is a cryptocurrency that has become a household name in recent years; it dominated news headlines on several occasions in 2017 as its price dramatically climbed and peaked at over $19,000 in December 2017, a 1,824% value increase since January of the same year [4]. What may not be as widely known is the excellent vehicle and tool Bitcoin can be for illegal activity. Bitcoin's relative lack of tractability, perceived anonymity and lack of regulatory oversight [5], compared to traditional fiat currencies, has rapidly attracted criminals such as illegal arms dealers, kidnappers, people smugglers, drug traffickers, blackmailers and terrorism financers [6] [7].

This unregulated nature of cryptocurrencies has been a cause of great concern for governing bodies; the Chinese government banned residents from trading cryptocurrencies in 2017, and the Bank of England's Governor Mark Carney has publicly expressed concerns about cryptocurrencies [1]. Recent studies have estimated that over a quarter of all Bitcoin users (26%) and nearly half of all Bitcoin transactions (46%) are associated with illegal activity [1]; in March 2017 46% of Bitcoin's transaction value equated to $76 billion, which is close to the scale of the US and European markets for illegal drugs [1]. Despite numerous 'darknet' marketplace seizures, such as the infamous Silk Road which was shut down by the FBI in 2014 who seized over $4 million worth of bitcoin [1], the amount of illegal activity associated with bitcoin remained close to it's all-time high as of April 2017 [1].

However, many cryptocurrencies have a distinct weakness for those wishing to evade the law; every transaction is published on the 'public ledger' that is the Blockchain and will remain forever accessible. Meanwhile, the capability of digital forensic tools advance further as research into de-anonymising Bitcoin activity continues. Right now, the individual identities of users may be masked by the pseudo-anonymity of their Bitcoin address, but through heuristics based on idioms of use, knowledge of wallet software and even analysis of user behaviour, Bitcoin addresses can be grouped into *clusters* [see 2.3] of addresses known to be controlled by a single user. Combine such clustering with a graphical, interactive representation of Blockchain data, and suddenly Bitcoin activity becomes more transparent and digital forensic investigations are made more accessible.

1.2 Contributions Outline

In this project we build such a graphical, interactive tool *Radar* that can be used to assist with digital forensic investigations across the Bitcoin Blockchain. The specific contributions of this project are:

- *Astrolabe*: In chapter 3, we develop an extension of the Blockchain Health project by Max Baylis which facilitates downloading the Bitcoin Blockchain and writing data in a CSV format suitable for import into a graph database.
- *Compass*: In chapter 4, we develop a tool for retrieving historical bitcoin exchange rate data in several fiat currencies.
- *Quadrant*: In chapter 5, we build a tool for retrieving mappings of wallets (entities) to the Bitcoin addresses known to be under their control from *walletexplorer* [1].

[1] http://www.walletexplorer.com

- *Balestilha*: In chapter 8 we take several different approaches to the implementation of a clustering algorithm using a *multi-input* heuristic [see 2.3].
- *Radar*: In chapter 9, we build a single page web-application which provides a graphical, interactive interface to Bitcoin's entire history; boasting near-instantaneous response times when navigating through Bitcoins 420+ million transactions [8] using intuitive graph controls. The tool provides various features to initiate and assist with a digital forensic investigation for Bitcoin.

Chapter 2

Background

2.1 Bitcoin

The model of traditional banking is based on a centralised trusted authority [9]; we use a third party that we trust in order to mediate the transfer of funds, a process in which we must explicitly define the payee and the payer (i.e. we must identify our friend and ourselves, if we were transferring money to a friend). Bitcoin has no central authority; it is a distributed peer-to-peer system. Therefore, sending money to our friend as before no longer requires any mediation; we send it directly to them. However, transferring funds peer-to-peer of course possesses several challenges; how do you prevent double spending? How do we ensure that money is not counterfeit? [9] How is currency issued without a central authority?

Bitcoin solves these challenges by relying on the *Blockchain* [see 2.1.4] for its 'source of truth', which itself is secured through *mining* [see 2.1.5] and network-wide consensus [9]. All *transactions* [see 2.1.8] are publicly available to view on the Blockchain and can be verified by any *node* [see 2.1.9].

2.1.1 Addresses, Keys & Hashing

A fundamental concept to Bitcoin is public key cryptography. The basic idea is that we generate a private and public key pair. We can pick a private key at random and use that to generate a public key (using elliptic-curve-crypto). The public key can then be used to receive funds [see transactions 2.1.8] and the private key used to sign transactions to spend the funds. [9]. It is critical to Bitcoin security that the process of generating a public key from a private key is one way and using a public key, it should be impossible to generate the private key.

A Bitcoin address can be generated from a public key. This Bitcoin address is the only information that a user needs to offer in order to receive payments, and since the address essentially appears as a random combination of letters and characters, it does not *directly* identify the user who generated the address. However, since all transactions are published publicly, Bitcoin only has pseudo-anonymity [see 2.2].

2.1.2 Bitcoin Address Types

A Bitcoin address may begin with a 1, a 3 or a bc1 which distinguishes between different types of addresses. A Bitcoin address beginning with 1 indicates a Pay-to-PubKey-Hash address. An

address beginning with 3 indicates a Pay-to-Script-Hash address. Addresses beginning with bc1 are a Bech32 type address (a segregated-witness address) [9].

2.1.3 Vanity Addresses

Vanity addresses typically begin with characters that contain human readable messages; for example, the address `1LoveBPzzD72PUXLzCkYAtGFYmK5vYNR33` is a vanity address as it begins with the message 'love' [9]. Vanity addresses are generated by generating and testing billions of candidate private keys, generating the corresponding public key and comparing the public key to the desired pattern until a match is found. Vanity addresses are typically used to create a more distinctive address, for example for businesses to reassure customers they are paying the correct address and to attempt to prevent adversaries substituting their own address to steal payments.

2.1.4 Blockchain

The Blockchain is the global ledger for Bitcoin. It contains the entire Bitcoin history, from the genesis to the most recently mined block. It exists as an ordered, back-linked list of blocks of transactions [9] where each block is linked to its previous block, known as its parent block, up to the genesis block.

A block contains a header, containing metadata, then a list of transactions that are included in that block [9] (see figure 2.1). A block can be uniquely identified by a cryptographic hash of its header. The metadata of the block contains data such as the previous block hash (used to create the chain link to the parent described earlier), a difficulty target and the nonce (used in the process of mining explained in 2.1.7).

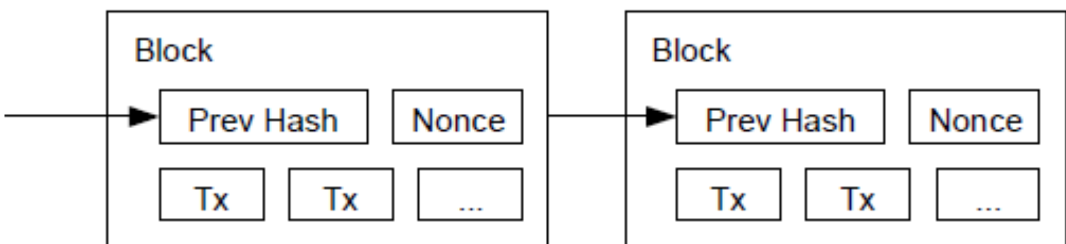

Figure 2.1: A simplified structure of a Bitcoin block. [9]

2.1.5 Mining

Mining is the mechanism that underpins the decentralised clearinghouse of transactions and also the mechanism in which new bitcoin is (currently) issued. Miners, incentivised by the reward of bitcoin, continuously compete to solve the *proof-of-work algorithm* [see 2.1.7] for each block.

As soon as a miner finds a solution for a block, they propagate this to the network, allowing each node to independently verify the block. In order to receive compensation for their efforts, a miner must include in the block a special transaction from the *coinbase* [see

2.1.6] to their own public address. If the block is valid, then the block will be added to the Blockchain, and eventually, the miner will be remunerated with newly minted bitcoin by the special *coinbase* [see 2.1.6] transaction. The successful miner also claims the transaction fees for the transactions included in the mined block. For all nodes that see this newly mined block, and is part of their longest chain, the next round of mining begins immediately, again competing to find the solution of the next blocks proof-of-work algorithm.

2.1.6 Coinbase

The coinbase refers to the input of a special type of transaction (the coinbase transaction) which, unlike regular transactions, does not consume bitcoin (i.e. it has no 'spending element'). Rather, it has a singular input called the *coinbase*. The bitcoin is therefore created out of nothing. This is the mechanism in which new bitcoin are introduced into the network.

2.1.7 Proof of work

New blocks are mined approximately every 10 minutes; this rate of mining is maintained even with fluctuations in the hash-rate of the network by periodically adjusting the difficulty of the proof-of-work algorithm. A change in hash-rate can be attributed to an increased/reduced amount of computational power of the network; this can be caused by a change in the number of participants or advancements in mining hardware.

As shown in figure 2.2 the hash-rate of Bitcoin has grown exponentially in recent years, reflecting an increase in the number of nodes contributing to the network in addition to advances in hardware. There is, however, a noticeable decline in hash-rate quite recently (around the time of November 2018) due to a *hard-fork* [see 2.1.12]. Comparing the hash-rate to difficulty over time (see figure 2.3) demonstrates how Bitcoin's difficulty is dynamically adjusted in response to changes in the networks hash-rate.

So, what exactly is the proof-of-work algorithm that miners are trying to solve? Firstly, observe in figure 2.1 there exists a 'nonce' field. This exists as a variable that can be changed freely by a miner. As simply put as possible, the miner wants to find a hash for the block that is lower than some target value. This can be achieved by repeatedly adjusting the nonce field and generating the new resulting block hash and comparing it to the target hash. Since a hash function is one-way, we have the property that a miner can only find a hash lower than the target is by performing repeated trial and error of nonces/hashes (i.e. iterating through different values for the nonce and looking at the hash of the block with that nonce value).

Clearly, the problem of finding a hash lower than a target can have its difficulty adjusted by adjusting how small or large the target is. A lower target will have fewer hashes that are smaller than it, and therefore, a lower probability of finding a satisfying hash; the difficulty and the target are inversely related.

In summary, the Proof-of-Work problem creates a measure of computational effort that must have been spent in order to achieve a valid solution. Although it is theoretically possible to find a satisfying solution immediately, it is incredibly unlikely. By adjusting the difficulty accordingly, Bitcoin achieves an average of 10 minutes' worth of the networks computational effort being spent for each solution that is found.

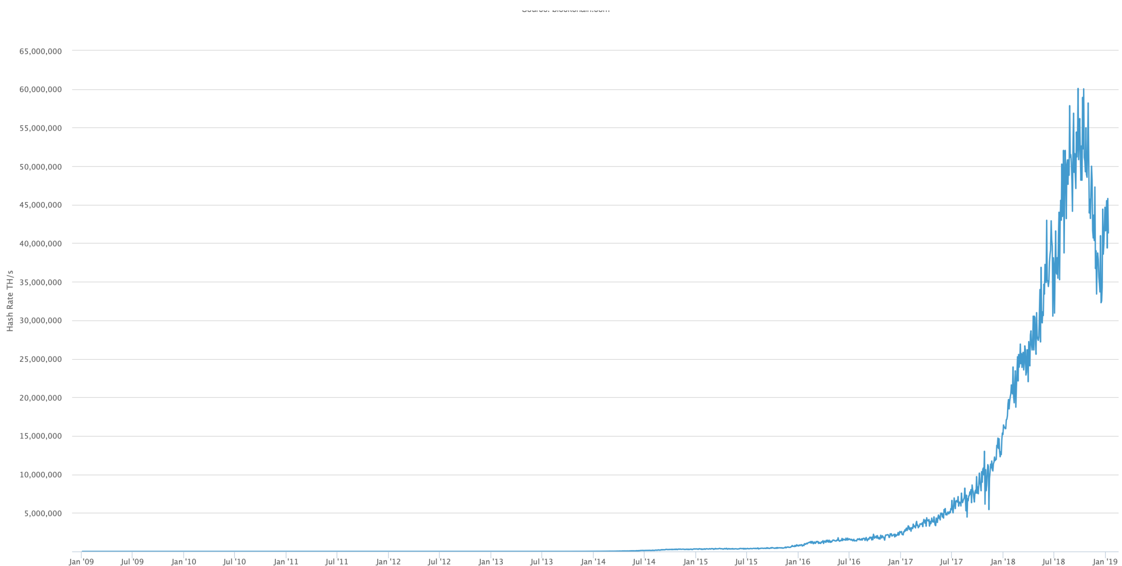

Figure 2.2: Bitcoin hash-rate for all of time [10].

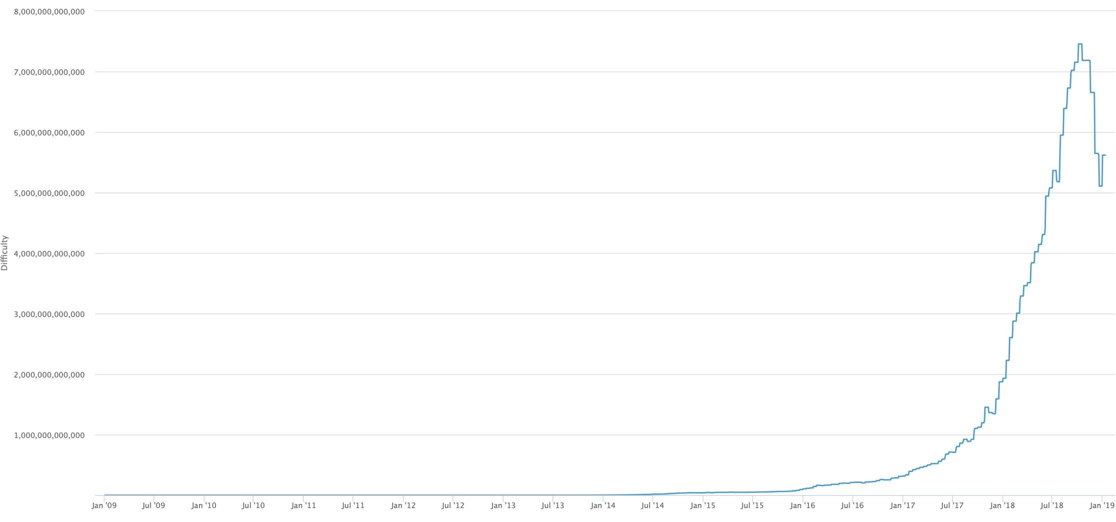

Figure 2.3: Bitcoin difficulty for all of time [11].

2.1.8 Transactions

Transactions are the core component facilitating the transfer of funds in Bitcoin. A transaction tells the network that the owner of some bitcoin has authorised the transfer of that bitcoin denomination to another owner [9]. In the simplest case, transactions contain inputs, which represent where the funds are coming from (the sender), and outputs, which represent where the funds are going to (the new owner).

For the transfer of funds to be confirmed, the transaction must be added to the global ledger for everyone to see. This means it must be included in a block that is mined on the

Blockchain. To be included in a future block, the transaction must be propagated to the many nodes of the network. The creator of the transaction must, therefore, send it to some of the Bitcoin nodes it knows the location of (its neighbours). These nodes, if the transaction is valid, will propagate the transaction to their neighbours, which will repeat the same process. This is the process of 'flooding' the network with the transaction. In addition to sending the bitcoin value to the recipient, the sender must also include an incentive for the miner to perform the work to include the transaction in the block they are currently mining. This is the *transaction fee*.

The transaction fee is a small value that is implicitly included in the transaction by leaving some left-over funds from the inputs once the recipient of the transaction has been 'paid'. This fee is collected by the miner who includes the transaction in the block they mined. A larger fee will act as a higher incentive for a miner to include the transaction in the block and will likely result in the transaction being included in the Blockchain in a timelier fashion.

2.1.9 Nodes

There are several different types of nodes in Bitcoin. One type of node is the 'full node'. Full nodes maintain a copy of the entire Blockchain locally, containing all transactions. These nodes must maintain and build their copy of the Blockchain by listening to incoming transactions and blocks. A full node can independently verify each transaction and block using its own copy of the Blockchain, without relying on information from other nodes. The disadvantage of running a full node is that it consumes a large amount of storage.

Not all nodes require a full copy of the Blockchain, and in many cases cannot hold a full copy due to resource constraints on devices such as smart-phones. These nodes are called Simplified Payment Verification (SPV)[9] nodes. These nodes do not have a full picture of the history of Bitcoin (i.e. all of the transactions) but do know all of the hashes of the blocks in the Blockchain. An SPV node, therefore, uses the depth of the block that the transaction is in to verify it, in comparison to the full node which will build a full database of unspent bitcoin from the transaction to the genesis block and verify it is not a double spend. The SPV node will wait until the transaction is in a block at a depth of at least 6, relying on the knowledge that other nodes have accepted the transaction as confirmation that the transaction is valid.

2.1.10 Immutable History

The Blockchain ledger becomes more and more immutable as time passes; this is because, in order to add a different block into the Blockchain, you must expend effort to re-do the proof-of-work for that block. Then, in order to change a block buried under other blocks, you must also expend the energy to re-do the proof-of-work for all of the blocks it is buried under (since you now require a different parent block hash field value and will then get a different hash for child blocks). This makes it exponentially less likely to be able to catch up with the main chain as the chain grows [12]. Therefore, the more blocks a block is 'buried' by, the more immutable it becomes, making old blocks in the chain are practically immutable [9].

2.1.11 Mining Pools

As touched upon earlier in mining [see 2.1.5], a miner must expend enormous amounts of computational effort in order to compete for the proof-of-work solution of a block, resulting

in them being compensated for their efforts with new bitcoin.

A miner without a substantial amount of resources at their disposal (i.e. their individual contribution to the network's hash-rate is near-negligible) will be unlikely to successfully mine a block and be compensated for their efforts. It may be the case that they must mine for many years before they are successful. This makes investing in hardware and energy a large gamble for many prospective miners; they will likely prefer smaller, but more frequent enumeration for their efforts. Mining pools are the solution to this problem for many miners.

Mining pools essentially combine many miners' resources in order to mine a block. Using many participants, a pool will have a much higher likelihood of being successful in mining a block. The reward of bitcoin is then paid to the pool, which the pool then distributes amongst the contributing members of the pool. A pool measures contributions of miners by giving the miners a much lower target for the block they are mining than the actual, more difficult, Bitcoin target. This will allow miners to mine blocks with the goal of finding a solution for the lower pool target, which the pool can recognise and use this as a measure of their contribution. Occasionally a pool miner will mine a block that meets the lower pool target and also meets the network's target; the pool will then propagate this solution to the network, claim the reward and distribute it according to miners contributions (taking a cut for itself, of course).

2.1.12 Forks

A fork on the Blockchain can occur naturally; an inherent property of a decentralised network, such as Bitcoin, is that different nodes can have different views of the world (i.e. due to transmission delays). However, these forks are usually quickly corrected within a small number of (usually one) blocks [9]. There also exist forks that have been deliberately created, i.e. by an attacker attempting to re-write the history of the Blockchain, or by a *hard-fork* software release.

A hard-fork in the Blockchain is where the Bitcoin network permanently diverges. A hard-fork occurs when there is a change in the consensus rules. The consensus rules tell a node which blocks to accept, and which ones to reject. A block must conform to the consensus rules of the majority of nodes in order to be added to the chain. When a change to consensus rules occurs, not all nodes may be on-board. Some may still be running with the old consensus rules. This would mean they would create and accept blocks using the old rules rather than the new ones - meaning their blocks will not be accepted by nodes running with the new rules (hard-forks mean a non-backwards compatible change). This leads to a divergence in the chain, where one chain is based on blocks added by nodes running the old rules, and the other chain containing blocks added by nodes running with the new rules.

In a similar vein, there also exist 'soft forks' which do not lead to a divergence of the Blockchain since they incorporate a backwards-compatible change. This means that blocks added by nodes running the old software will not be rejected by those running the new software.

Figure 2.4 shows how a hard-fork will look on the Blockchain; the fork at block 3 represents a naturally occurring fork, as mentioned at the beginning of this section, whereas the fork occurring at block 6 is a hard-fork and may be due to a change in the consensus rules of

the network [9].

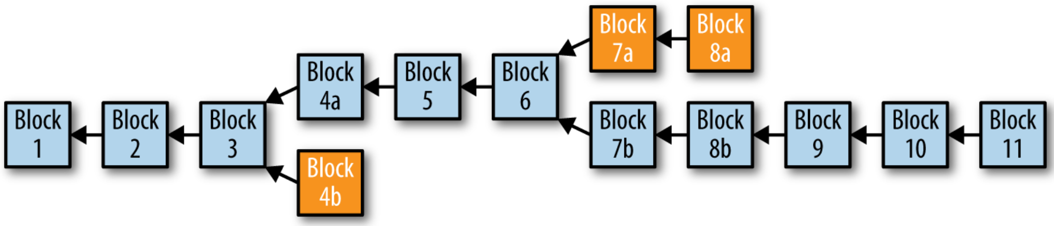

Figure 2.4: An example of a hard-fork [9].

2.1.13 Bitcoind

Bitcoind (Bitcoin Core daemon) is a headless daemon which implements the Bitcoin protocol for RPC use.

2.2 Anonymity

Since all transactions to occur in Bitcoin are available to view by the public, Bitcoin can only offer pseudo-anonymity rather than real anonymity [13]. There exist studies which show it is possible to de-anonymise Bitcoin transactions based on data that is publicly available [13]. Therefore, there exist services called 'mixers' which aim to obfuscate transaction origins with the goal of strengthening privacy and make such de-anonymisation more challenging.

2.2.1 Mixing Services

A mixing service, also known as 'mixer' or 'tumbler', works in the following way: A user wishing to use a mixing service will first create a new address and send bitcoin to an address of the service, asking the service to send the funds back to their new address. Other users also using the mixing service will take the same steps and send bitcoin to the service. The service now holds bitcoin for multiple users. The service can now use any of the addresses in which it holds bitcoin to send money back to the users of the service. This results in the appearance of a disconnect between the user's old address (the one which held the bitcoin initially) and the new address which now holds their bitcoin [14]. Clearly, this helps disguise the origins of bitcoin and can be used as a tool in the process of money laundering. The operators of these services profit by charging a fee in exchange for 'mixing' their bitcoin.

2.2.2 Risks of Using Transaction Anonymisers

In order for a mixing service to provide functionality, it will likely retain a history of the senders and recipients. If an attacker wishing to discover users using these services were to set up a Mixer, they could potentially gain full knowledge of relationships between senders and recipients if logs were to be kept for this data over a large enough time span [14]. Of course, this would only be effective if the user relies on a single mixing service; a user could mitigate this vulnerability by using multiple mixing services, but at the cost of paying more

fees.

Other weaknesses in anonymisation services could be the timing of incoming and outgoing transactions, in addition to transaction values, which could all be used to correlate the senders and recipients of bitcoin. Furthermore, the communication between the user and the service itself, if compromised, could reveal information (such as addresses) used to de-anonymise transactions.

In addition, studies have shown some other mixing services such as Bitlaunder, DarkLaunder and Coinmixer to have multiple serious security flaws and can be easily exploited to compromise the privacy of those that use it. In fact, the investigation shows that making a genuinely secure mixer is a difficult task, which may be refreshing news for law enforcement wishing to taint bitcoin back to its source, but worry-some for legitimate users of such mixing services [15].

2.2.3 Peeling Chain

A peeling chain is a pattern of use that exists widely in the Bitcoin network; it can be used in withdrawals from exchange services and mining pools, and in some cases, it forms part of a signature of illegal activity. The chain begins at an address that often holds a large bitcoin value, and the goal is to obfuscate the funds that a wallet holds. This is achieved by a series of transactions in which the bitcoin will be sent to two or more addresses, one of which will belong to the service (which owns the originating address) where the majority of the bitcoin will be sent, and the small remainder to some change addresses.

Recently, peel chains are being used less frequently due to modern Blockchain analysis software developing the capability to collapse even the most complex peel chains. There also exists transaction fees at each step of the peel chain, which may make them economically inefficient.

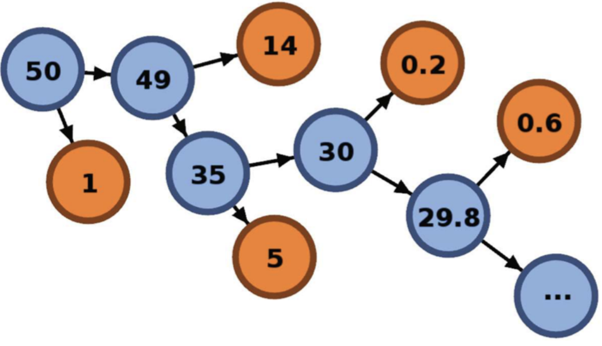

Figure 2.5: An example of a peel chain. [15]

2.2.4 Taint Analysis

Taint analysis is a measurement of the link a denomination of cryptocurrency has with previous illegal activity. For instance, if a vendor has accepted a payment of bitcoin where

one of the 3 inputs to the transaction was stolen in a theft, some part of the output of that transaction will have a 'taint' measurement to show its link to the theft, even though the vendor in this example knows nothing of the theft. Taint analysis, therefore, impacts the fungibility of bitcoin.

Taint analysis was offered in a feature for free by blockchain.info. The definition of 'taint analysis' on the site was 'Taint is the % of funds received by an address that can be traced back to another address'. The taint therefore usually correlates with the percentage of funds that are linked to some theft of coins or are known to have been used in some illicit manner. However, this feature is no longer available on the site, with some speculation to the feature being retracted in order to provide it as a charged, premium offering.

2.2.5 TOR

The onion router (TOR) is network infrastructure used to obscure geographical locations of IP addresses by routing communications through multiple proxies located around the globe. Messages are sent in multiple layers of encryption for each relay, so each relay can decrypt the outer layer and forward the decrypted resulting message onto the next router. Eventually, the one-layer encrypted message will reach its destination. Using Tor therefore permits for protection against eavesdropping and traffic analysis [16].

2.3 Bitcoin Address Clustering (on-chain)

It is possible to define a number of heuristics that can be used in an effort to group distinct Bitcoin addresses so that they can be collectively associated with an individual user. This section outlines some of these heuristics that use data from the Blockchain to help cluster addresses belonging to the same user.

2.3.1 Multi-Input Transactions

Multi-input transactions will be required when some user, say Alice, wishes to transfer some funds, say the value of v to some other user Bob; however, Alice does not hold a single bitcoin denomination that is greater or equal to v and therefore must combine multiple smaller denominations to meet/exceed the value of v.

It is, therefore, a safe assumption to say the owners of each input of a transaction belong to the same user, regardless of the distinct addresses each input is locked to [17]. This heuristic, therefore, leverages this assumption to cluster together addresses whenever they are the inputs of the same transaction.

2.3.2 Change Addresses

A transaction will often generate change when the value of the inputs exceeds the amount to be paid to the recipient and any transaction fees, and the remainder will want to be paid back to the sender. To do this, Bitcoin will generate a change address for the remainder to be sent to. Therefore, it can usually be safely assumed that when a transaction has two outputs, and one is a new address that has not appeared before, that this address is the change address and belongs to the sender of bitcoin for this transaction [17].

A weakness in using this heuristic as highlighted in previous research [18] is that it is not robust in the face of changing patterns of use in the network since it is an idiom of use rather than an inherent property of Bitcoin. In fact, a study by Sarah Meiklejohn et al. [18] found that falsely linking just a small number of change addresses causes entire relationship graphs to collapse into giant clusters that are not actually controlled by a single user.

However, through careful investigation of false positives and implementing a more conservative clustering algorithm, it was possible to name 1,600 times more addresses post-clustering than those already identified through manual tagging [18]. Evidently, these heuristics could be extremely useful in the process of identifying users and tracking funds, such as by collapsing peeling chains [see peeling chain 2.2.3].

A more conservative algorithm uses additional measures for greater robustness of the algorithm. **Robustness:** For each transaction, if multiple outputs meet the pattern of a change address, no address is labelled as the change address. A change address is labelled iff exactly one output meets the pattern.

- Avoid self-change addresses, where the change address is specified as the input address.
- The address does not appear in any other transaction.
- The transaction is not a coin generation transaction.
- All other output addresses have appeared in previous transactions.

2.3.3 Consumer Wallet Heuristic

This heuristic as presented in 'Data-Driven De-Anonymization in Bitcoin' by Jonas David Nick [19] uses the assumption that popular consumer wallets (such as Bitcoin Core, Electrum, MultiBit, Armory, Android Bitcoin Wallet, etc) by default only allows users to send bitcoins to a single address. Therefore, transactions generated by a consumer wallet will only have one or two outputs; one to the recipient address and one as the change address (if change was required).

This heuristic can then be used in clustering by identifying change addresses: for every public address p, find the transactions ts in which an output locked to p is spent (i.e. p spends bitcoin) then ensure every t in ts has less than 3 outputs. If p is spent by a transaction that has more than 2 outputs, it is not a change address [19] .

2.3.4 Optimal Change Heuristic

This heuristic is also presented in 'Data-Driven De-Anonymization in Bitcoin' by Jonas David Nick [19]. It relies on the assumption that wallet software does not spend unnecessary outputs when constructing transactions, since including more outputs will necessary will lead to unnecessary bloat in the size of the transaction and therefore higher transaction fees.

Using this assumption, you can assume that the change value will be smaller than any of the input values; since if this wasn't true and it was larger, the smaller input could just be omitted from the transaction and the change output value will be reduced by this amount.

Therefore, if a transaction has a unique output which is smaller than all inputs, it is very likely to be the true 'optimal change output' [19].

2.3.5 Behaviour Based Analysis

Humans naturally fall exhibit behaviour patterns, and since many events on the Blockchain are human-driven, it is possible to attempt to identify these behaviour patterns and apply them in clustering the activities of distinct users. Using timestamps and network properties, it becomes possible to observe such behaviour patterns could include items being purchased, daily schedule and activities, in addition to non-human behaviour patterns of hardware and network latencies [20].

The study 'Identifying Bitcoin users by transaction behaviour' [20] shows that a transaction can be described by its timestamp, connectivity (number of inputs/outputs) and coin flow. Features can be extracted that help characterise some aspect of transaction behaviour over time. These features are:

- The time interval between successive transactions
- The hour of the day the transaction took place
- The time of hour (seconds since the start of the hour)
- The time of day (seconds since the start of the day)
- Coin flow - the net bitcoin value
- Input/output balance - The balance of inputs from other users compared to outputs going to other users.

2.4 Bitcoin address clustering (off-chain)

The previous section outlines heuristics for using on-chain data for address clustering, however, it is possible that there exists more public information available elsewhere on the internet that can be used to help cluster addresses [21].

2.4.1 Tag Collection

Tag collection is the process of trying to find a Bitcoin address that is mentioned in the same data frame as some tag (such as a username or a company name) [21]. Therefore, passive tag collection can be carried out by crawling sites where this information is likely to appear, such as social media, bitcoin forums and dark-web marketplaces (for example, Silkroad). Tagging in this manner may allow addresses to be correlated to known Bitcoin businesses, or categorised in the type of service they are, such as an exchange, marketplace, mining pool, mixer, gambling etc.

2.4.2 Entity Clustering

Sarah Meiklejohn et al.'s work [18] is an example of how addresses can be clustered to be under the control of known entity by proactively engaging with the services through transactions, which require a public address of the entity. This provides the minimal 'ground

truth' data needed to bootstrap the formation of larger clusters using the on-chain heuristics in section 2.3.

2.5 Popular Services

2.5.1 Satoshi Dice

A very popular Bitcoin dice game, introduced in April 2012, is Satoshi Dice. Users may place bets and, if they win, have some multiple of their bets paid back to them [18]. It is estimated that Satoshi Dice is responsible for about 60% of activity on the Bitcoin network and is expected to contribute an extra 14MB to the overall Blockchain daily [18].

2.5.2 Exchanges

Often it is unavoidable to use an exchange service. Exchange services enable a user to exchange their currency, such as a fiat-money or another cryptocurrency, into bitcoin and vice versa. For those users attempting to user bitcoin for illicit means, this poses an issue as it is a point of centralisation; an example is a user who has stolen bitcoin, who wishes to then convert stolen funds to their fiat currency, but first must go through a known exchange. It's possible their bitcoin is now tainted [see 2.2.4] and will not be accepted by many exchanges.

2.6 Illegal Activity

The impact Bitcoin can have in facilitating illegal activity is substantial; in a 2012 intelligence report [22], the FBI claimed:

> "If Bitcoin stabilises and grows in popularity, it will become an increasingly useful tool for various illegal activities beyond the cyber realm. For instance, child pornography and Internet gambling are illegal activities already taking place on the internet which require simple payment transfers. Bitcoin might logically attract money launderers, human traffickers, terrorists, and other criminals who avoid traditional financial systems by using the Internet to conduct global monetary transfers."

The report now 7 years ago feels almost like an early warning call; since then we have seen the FBI's predictions materialise as we see high-profile thefts, money laundering and drug-buying stories hit headlines.

Why does Bitcoin make life difficult for law enforcement? Bitcoins decentralised nature introduces a number of vulnerabilities to traditional law enforcement techniques. There is a lack of anti-money-laundering software, account owners often cannot be directly identified, and it is more difficult to identify the original source of funds [22].

2.6.1 Silk Road

Silk road was an international online marketplace which operates as a Tor hidden service that was overwhelmingly used as a market for controlled substances and narcotics [23]. Through investigation by Nicolas Christin [23], it appeared to be quite an advanced marketplace; including features for a product rating system with customer feedback, an escrow account for

dispute management, automatic price pegging to the USD for sellers (accounting for bitcoin price fluctuations) and site-wide promotional campaigns such as 'pot day' promoting the sale of cannabis on the site [23]. However, in 2013 the FBI shut down Silk Road, but an almost identical site Silk Road 2.0 quickly emerged and began generating sales of $8 million a month with 150,000 active users. Just a year later, the FBI also shut down Silk Road 2.0 and arrested it's operator [24].

2.6.2 Money Laundering

Bitcoin can be used for the purpose of money laundering; funds routed through *mixers* [see 2.2.1] can conceal the origin and destination of funds such that it can be traced by law enforcement. This could ultimately facilitate perpetrators in the act of concealing or mischaracterizing the proceeds of crime (a.k.a money laundering).

2.6.3 Bitcoin ATM's

Bitcoin ATM's are physical machines, often installed in small shops, that allow users to buy and sell various cryptocurrencies by trading in their fiat-money. For example, there exist many Bitcoin ATM's in London, UK, where you buy bitcoin in exchange for depositing pound sterling. The owners of these machines have the incentive of hosting them in their stores as they can charge a commission fee on these transactions. Over the last few years, these ATM's have become increasingly abundant across the UK, as shown in figure 2.6.

However, there have been growing concerns that these ATM's are being used in the process of money-laundering. In a December 2018 report by Bloomberg [25], reporters carry out an experimental investigation into the anti-money laundering measures being taken by Bitcoin ATM's; estimating that more than half of the machines in the US are not following the rules. Many machines do not verify identification or impose limits on transactions, in direct violation of several US banking laws [25].

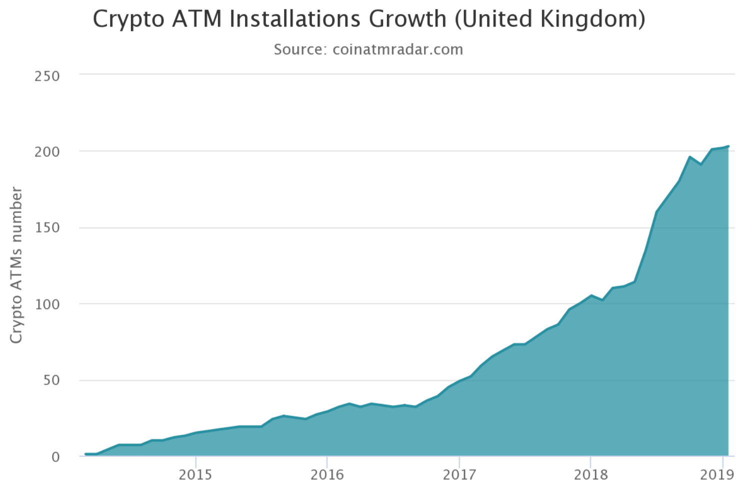

Figure 2.6: Accumulated number of crypto ATMs installed over time - UK only [1]

[1] https://coinatmradar.com/charts/growth/united-kingdom/

2.6.4 Significant Thefts

Not only is Bitcoin used as a mechanism to facilitate off-chain illegal activity; crime also occurs on-chain in the form of Bitcoin thefts. Thefts are quite common in Bitcoin, and those of significant value often make the headlines. Below are some major thefts [2]:

Theft Name	**Theft Victims**	**Approx Loss (฿)**	**Theft date**
Mass MyBitcoin Thefts	MyBitcoin users with weak account passwords	4019	June 20th-21st 2011
MyBitcoin Theft	MyBitcoin & customers	78739	July 2011
Bitcash.cz Hack	Bitcash.cz	484	November 11th, 2013
Linode Hacks	Bitcoinica, Bitcoin.cz mining pool, Bitcoin Faucet, others...	Bitcoinica: 43554, Bitcoin.cz: 3094, Bitcoin Faucet: 4	March 1st-2nd 2012
Bitcoinica Hack	Bitcoinica	18547	May 12, 2012
Bitcoinica Theft	Bitcoinica	40000	July 13th, 2012
CryptoRush Theft	CryptoRush	950	March 11th, 2014

The thefts above were selected out of an extensive collection since these thefts, in particular, include victims of entities collected in the entity tagging section of this project 5.

2.7 Existing Forensic Tools

There exist several digital forensic tools, both free and proprietary; we present in this section the information on these tools made available to the public (i.e. without making a purchase).

2.7.1 Blockchain Explorer

Blockchain Explorer, accessible at https://www.blockchain.com/explorer, provides a wealth of information (available on the Blockchain) about addresses, transactions and blocks. For instance, users can search by a public address in order to inspect the transactions associated with it. It also has features to provide charts on Bitcoin statistics, such as the network hash rate and exchange rates. However, in order to carry out a forensic investigation, it will often be necessary to follow the path of funds between addresses, which would be simplified if it were possible to do this graphically; Blockchain Explorer does not provide a graphical interface.

[2] https://bitcointalk.org/index.php?topic=576337

[3] https://www.blockchain.com/explorer

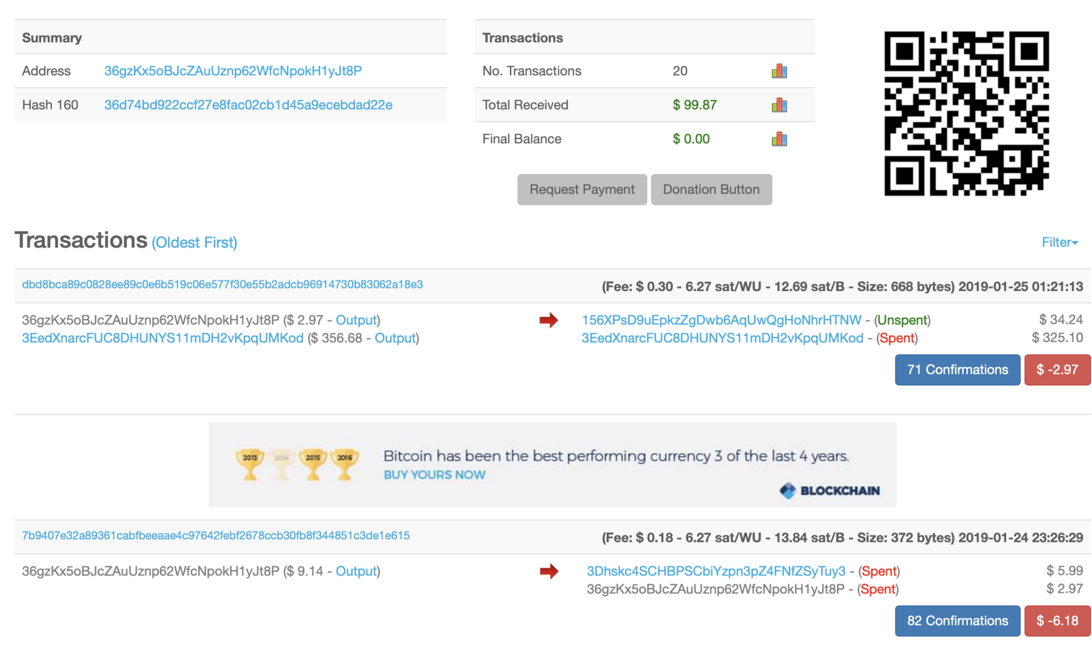

Figure 2.7: Blockchain Explorer's address investigation tool [3]

2.7.2 Chainanalysis

Chainanalysis is a company which have built a proprietary software solution for digital blockchain forensics. We cannot know exactly the solutions Chainanalysis provide, or how, other than through the information used to advertise their product.

ChainAnalysis seem to have 3 main categories of customers and describes features they can offer to target each type of customer individually. The types of customers and their associated features are:

- Financial Institutions: Focus on meeting Anti-money laundering (AML) and Know Your Customer compliance obligations. Tools for due-diligence and detecting suspicious activity.
- Cryptocurrency Exchanges: Focus on AML obligations and due-diligence (similar to financial institutions offerings)
- Government: Features for suspect identification, criminal revenues and machine-learning based pattern recognition.

2.7.3 Wallet Explorer

Wallet Explorer is a website which provides similar capabilities to Blockchain Explorer in terms of inspecting addresses individually. However, Wallet Explorer additionally has a list of known entities and the public addresses they are known to be mapped to. This is extremely useful information in understanding patterns of use and the main players in the bitcoin

network. However, this data is represented in quite a difficult to use format; it is not linked with network activity and therefore, on its own, it is not very useful in carrying out forensic investigations.

Top wallets

Exchanges:	Pools:	Services/others:	Gambling:	Old/historic:
Bittrex.com	BTCCPool	Xapo.com	SatoshiDice.com (original)	AgoraMarket
Huobi.com (2)	SlushPool.com (old) (old2)	Cubits.com	LuckyB.it (chatbot)	BetcoinDice.tm
Poloniex.com	GHash.io	CoinPayments.net	BitZillions.com	SilkRoadMarketplace
BTC-e.com (output) (old)	AntPool.com (old) (old2)	BitPay.com (old) (old2) (old3)	999Dice.com	DeepBit.net
Luno.com	BitMinter.com	Cryptonator.com (old)	CoinGaming.io	SilkRoad2Market
LocalBitcoins.com (old)	EclipseMC.com (old) (old2) (old3)	BitoEX.com	PrimeDice.com (old) (old2) (old3) (old4)	EvolutionMarket
Bitstamp.net (old)	KnCMiner.com	Cryptopay.me (old)	CloudBet.com	Instawallet.org
Cryptsy.com (old)	Bitfury.org	HaoBTC.com	SatoshiMines.com	UpDown.BT
Bitcoin.de (old)	BW.com	AlphaBayMarket (old)	NitrogenSports.eu	AbraxasMarket
Cex.io	Eligius.st	NucleusMarket	SecondsTrade.com	MintPal.com
BtcTrade.com	Kano.is (old)	BitcoinFog	PocketDice.io	SealsWithClubs.eu
YoBit.net	Telco214	CoinJar.com	FortuneJack.com	PandoraOpenMarket
OKCoin.com (2)		HelixMixer (old) (old2) (old3) (old4) (old5) (old6) (old7) (old8) (old9) (old10) (old11) (old12) (old13) (old14) (old15) (old16) (old17) (old18) (old19) (old20) (old21) (old22) (old23) (old24) (old25) (old26) (old27) (old28) (old29) (old30) (old31) (old32) (old33) (old34)	Rollin.io	MiddleEarthMarketplace
BTCC.com (old) (old2)		HolyTransaction.com	BitZino.com	BtcDice.com
BX.in.th		BTCJam.com (old) (old2)	BitcoinVideoCasino.com (old) (old2)	McxNOW.com
HitBtc.com (old)		CoinKite.com	Betcoin.ag (old)	SheepMarketplace
Kraken.com		MoonBit.co.in	SatoshiBet.com	DiceOnCrack.com
MaiCoin.com		BitcoinWallet.com	SafeDice.com	BlackBankMarket
MercadoBitcoin.com.br		FaucetBOX.com	Coinroll.com	BTCGuild.com
Bter.com (old) (old2) (old3) (cold)		OkLink.com	YABTCL.com	Coin-Swap.net
Hashnest.com		Purse.io	Crypto-Games.net	BlueSkyMarketplace
BitBay.net		ePay.info	Betcoin.tm	Justcoin.com
AnxPro.com		Loanbase.com	SwCPoker.eu	PinballCoin.com
Bitfinex.com (old) (old2)		GermanPlazaMarket	SatoshiRoulette.com	Inputs.io
Bleutrade.com		Paymium.com	BTCOracle.com	BitAces.me (old)
CoinSpot.com.au		Bitbond.com	Peerbet.org	AllCoin.com
Matbea.com		StrongCoin.com-fee	AnoniBet.com	Bitcoin-24.com (old) (old-hotwallet)
Bit-x.com		CryptoStocks.com	Satoshi-Karoshi.com (old)	Betcoins.net
Paxful.com		CoinApult.com (old)	777Coin.com	Bitcoin-Roulette.com
VirWoX.com		Genesis-Mining.com	BitStarz.com	Bitmit.net
BitBargain.co.uk		ChangeTip.com	SatoshiCircle.com	Cryptorush.in
SpectroCoin.com		DoctorDMarket	Coinichiwa.com	Leancy.com
Cavirtex.com		GoCelery.com	CoinRoyale.com (old) (old2)	Coin.mx
C-Cex.com (old)			BetMoose.com	Crypto-Trade.com
CoinHako.com			JetWin.com	VaultOfSatoshi.com
FoxBit.com.br (2) (cold) (cold-old)			BetChain.com-old	BitElfin.com
Vircurex.com			BitcoinPokerTables.com	ActionCrypto.com
BitVC.com			DiceNow.com	50BTC.com (old) (old2) (old3)

Figure 2.8: Wallet Explorers catalogue of wallet addresses [4]

2.7.4 Blockpath

Blockpath is a website that claims to be a Bitcoin accounting tool. The feature of this site that was the most interesting is the graphical explorer. The graph allows you to explore the relationship between addresses. However, there does not appear to be any mapping of multiple public addresses to a single entity (i.e. clustering), which would be vital to gaining real insights when performing forensic analysis.

[4] https://www.walletexplorer.com/
[5] blockpath.com

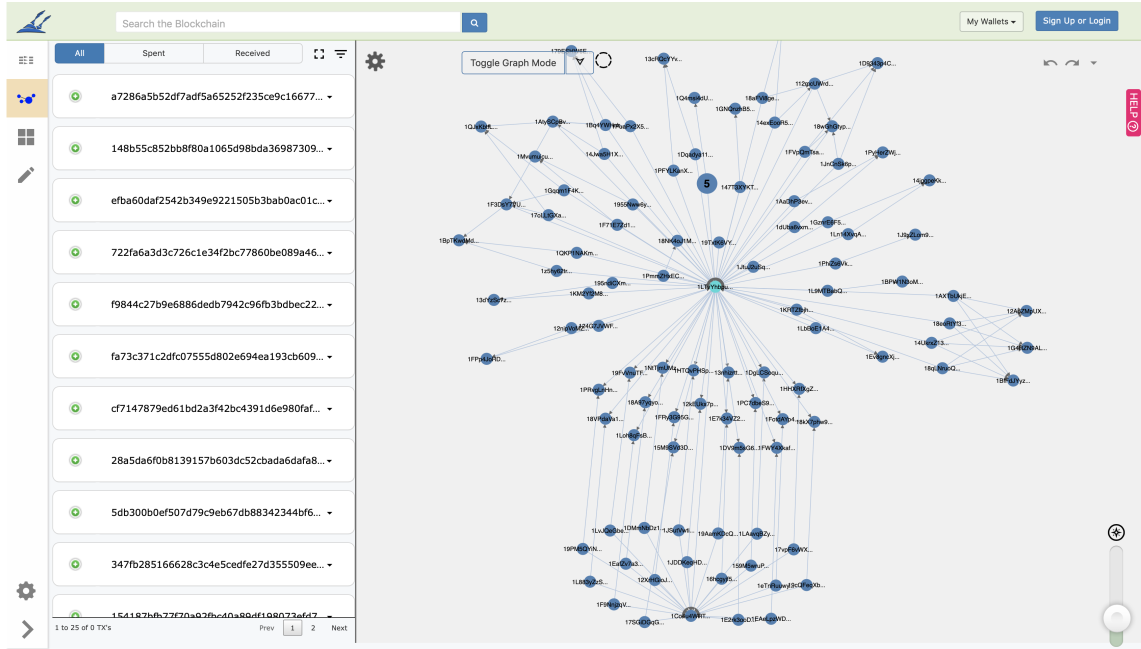

Figure 2.9: Blockpath's graphical explorer displaying transactions between addresses [5]

2.7.5 Other Solutions

There exist a number of companies claiming to offer similar solutions to Bitcoin analytics.

- **Elliptic** Similar to Chainanalysis, Elliptic offer proprietary software with main clients in finance and law enforcement. [See https://www.elliptic.co/what-we-do/bitcoin-forensics].
- **ScoreChain** Another proprietary software solution, claiming to perform Bitcoin analytics for compliance, forensics and CRM. [See https://bitcoin.scorechain.com].
- **Block Explorer:** Similar to Blockchain Explorer; providing visibility to information associated with single addresses and blocks. Also incorporates market information for many other cryptocurrencies and a cryptocurrency related news feature. [See https://blockexplorer.com/].

2.8 Importing Blockchain Data

There exists previous work in the domain of downloading the Bitcoin Blockchain. We first researched and assessed these implementations.

2.8.1 Bitcoin to Neo4J Tool: Open Source Project

There exists an open source tool, built by the author of the website `learnmeabitcoin.com`, which populates a Neo4J database with the entire Bitcoin Blockchain [26]. This tool requires a full Bitcoin node to be run in order to have the .dat files stored locally; the tool will parse the .dat files and write them, using Cypher queries, to the Neo4J database. This approach has the advantage that it is well respected in the community and has been cited a number

of times by those seeking to achieve the same goal [27]; however, the tool will take several weeks (apparently 60+ days) to complete the import.

2.8.2 Max Baylis: Imperial MSc Project 2018

Max undertook a project titled 'Blockchain Data Analytics and Health Monitoring' within the Department of Computing at Imperial [28] which involved bulk extracting data from several blockchains and writing that data to a MongoDB database. He additionally used Kafka for streaming new blockchain updates and writing those updates to the database. This data was used for providing blockchain analytics data in a web-based environment [28].

2.8.3 TokenAnalyst: Medium Blog

TokenAnalyst published a blog article on Medium with the title 'How to load Bitcoin to Neo4J in a day' [27]. They describe their process of using RPC to fetch the data, writing the data to a 'data lake' in a compressed Arvo format, generating CSV's from the compressed files and feeding them to Neo4J's bulk import tool. Similar to Max's work, they then use Kafka for steaming the most recent Bitcoin data to a tool which writes it to Neo4J to keep the database up to date.

2.8.4 Blockchain2graph: Open Source Project

A company Blockchain Inspector who are 'using Artificial Intelligence to fight fraud in the Blockchain' have open-sourced a tool they use with the claim that it extracts Bitcoin data and writes it to a Neo4J database [29].

2.8.5 Analysis of Previous Work

TokenAnalyst's approach seems to best describe the ideal situation for this project; an efficient bulk import into Neo4J and a mechanism for keeping the database up to date. There exists no open source implementation for their work, only a high-level description of what they did in their blog post. Max's work, however, is available to me and his work in fetching data using RPC could prove useful to our project. The Bitcoin to Neo4J tool would not be a feasible approach and can be immediately ruled out due to the time requirement of several weeks for the tool to complete; this would not be acceptable for the timescale of this project. As for Blockchain Inspector's open-source solution, upon inspection of the code, it seems the bulk import process relies on an existing database containing the entire Bitcoin Blockchain and therefore seems to be more of a tool to migrate from a traditional database to a relational one (Neo4J). It is therefore not very applicable to our work, creating an intermediate database storing the Bitcoin Blockchain for the bulk import would be unnecessary and expensive.

2.9 Know Your Customer

Know Your Customer (KYC) is a keystone in the fight against money laundering. It is the process of a business carrying out customer due-diligence measures and verifies the identity of its clients. This ensures the business is dealing with bonafide customers and organisations and helps identify suspicious behaviours or practices. These measures help satisfy anti-money laundering requirements (AML).

2.10 Privacy Enhanced Cryptocurrencies

2.10.1 ZCash

ZCash has shielded transactions in which a zero-knowledge proof is used to verify the sender, recipient and amount of a transaction, without exposing the information on the public Blockchain. Shielded transactions are encrypted on the Blockchain, yet still verified by the network using zk-SNAKRK proofs.

2.10.2 MimbleWimble (Protocol)

MimbleWimble is a blockchain protocol which addresses gaps in many blockchain protocols by using strong cryptographic primitives to provide good scalability, privacy and fungibility. Two current implementations of the MimbleWimble Protocol are Grin and Beam. Grin is developed in Rust and is a community-backed project.

2.10.3 Dash

Dash is a cryptocurrency which offers a privateSend feature where users can mix the funds they are sending with others on the network; making it more difficult for a third party to identify where funds came from. There exist master nodes on the network that conduct coin mixing.

2.10.4 Monero

Monero employs different privacy technologies to other crypto-currencies, such as Bitcoin and Ethereumm, which have transparent blockchains. Privacy-preserving technologies, such as *ring signatures*, *ring confidential transactions* and *stealth addresses*, enable a verifiable blockchain without exposing details such as the sender, recipient or the amount of a transaction. These privacy features all exist by design for all transactions, such that all transactions made with Monero are made private through these features, rather than the selective privacy provided by some other privacy enhanced crypto-currencies such as Zcash.

Kovri

Kovri is a private overlay network using routing and encryption, allowing users to hide their geographical location and IP address. This avoids the need to use Tor which has semi-trusted authorities, who could have an overreaching influence of network consensus (and therefore in the determination of who can relay traffic). Kovri makes passive surveillance impossible.

Stealth Addresses

Used to obscure the recipient address of a transaction. Stealth addresses are one-time (ephemeral) public keys. Observers cannot infer the recipient from the stealth address; however, the sender can verify that the payment was sent by then. A Monero wallet has a public view key and a public send key. When Alice sends Monero to Bob, Alice will use Bob's public view and send key, in addition to some random data in order to generate the one-time ephemeral key (stealth address). The entire Blockchain can see the stealth address. Bob, with his wallet's private view key, can identify the output of the transaction sent to him on

the Blockchain. Bob will be able to generate a one-time private key that corresponds to the one-time public key in order to spend the funds.

Ring Signatures

Used to obscure the sender. A group of possible signers are fused together to produce a possible signature. The actual signer is a one time spend key that corresponds with an output being spent from the senders' wallet; the non-signers in the ring are past transaction outputs pulled from the Blockchain being used as decoys. These outputs combined make up the inputs of a transaction in which it is indistinguishable between the actual signer and the decoy signers. Key imaged are used to detect double-spends.

Ring Confidential Transaction (CT) Signatures

Obscures the amount of Monero sent in a transaction. Before Ring CT's, Monero would require transaction amounts to be split up into smaller denominations and split amongst ring signatures to ensure there would be enough ring signers (since all signers of a ring must have outputs of the same value). This means an observer can see the amounts sent in transactions. Ring CT transactions obscure the value of outputs, but now the sender must commit to the value of an output (Pedersen commitment scheme) but without publicly disclosing the amount being sent.

2.11 Technology

2.11.1 Spring WebFlux

Spring Webflux is a reactive-stack web-framework which supports reactive data streams. Reactive programming is a paradigm built around the publisher-subscriber pattern which promotes asynchronous, non-blocking approach to data processing.

Key concepts of a reactive stream include *backpressure*, which is used to control the ingestion rate of data by a consumer to prevent consumers being overwhelmed with more data they can handle. The framework simplifies the processing of unbounded streams of data without incurring the overhead of buffers.

Chapter 3

Blockchain Download: *Astrolabe*

This section will cover the approach taken to retrieve and store all historical Bitcoin transactions, up to an appropriate (recent) block height of 570,000 - mined on 3rd April 2019.

3.1 Hardware

We had access to a VM on a machine, *Satoshi*, provided by the Department of Computing at Imperial. The VM has the following resource allocated to it:

- **Processor**: 16 of 24 cores, AMD EPYC 7401
- **Memory**: 16GB
- **Storage**: ~5TB SSD Available

3.2 Retrieving Historical Bitcoin Transactions

After analysing the various approaches taken previously to retrieve all bitcoin transactions, we decided to adapt the contribution of Max Baylis' Health Monitoring project [see 2.8.2] to build *Astrolabe*. Max's approach uses WebFlux [see 2.11.1] to create parallel streams of data, fetched using RPC, and writes them to MongoDB; we adapted this work to divert the data into a CSV format suitable for importing into Neo4J. Below we describe the steps taken to build CSV files for the entire Bitcoin Blockchain successfully.

- Introduced a new API endpoint. The new endpoint was named `extractBitcoinToNeo4j`. This accepted two arguments as path variables, `fromHeight` and `toHeight` which are to be used to define the range of blocks to fetch data from.
- Using the parameters `fromHeight` and `toHeight`, we then generate a Flux stream which uses RPC to invoke the method `getBlockHash` on a *bitcoind* [see 2.1.13 instance running in a container on the same machine.
- The Flux stream of block hashes are then mapped to the actual block data using Flux's `flatMap` operation and by invoking the method `getBlock`, passing it the block hash from the previous step.
- The block data is retrieved and deserialised to an intermediary representation in Java. Each block is then appended to a CSV file in the format required for an import into Neo4J.

- The retrieval of a single block will then initiate the process of fetching transactions; the Flux of blocks will be mapped to individual transactions by fetching each of the transaction ids in the block and again using RPC to invoke the `getrawtransaction` on the *bitcoind* instance in order to fetch all the data for each transaction.
- Each retrieved transaction will be written to CSV, in addition to writing the relationships between transactions, blocks and outputs to their own CSV files.

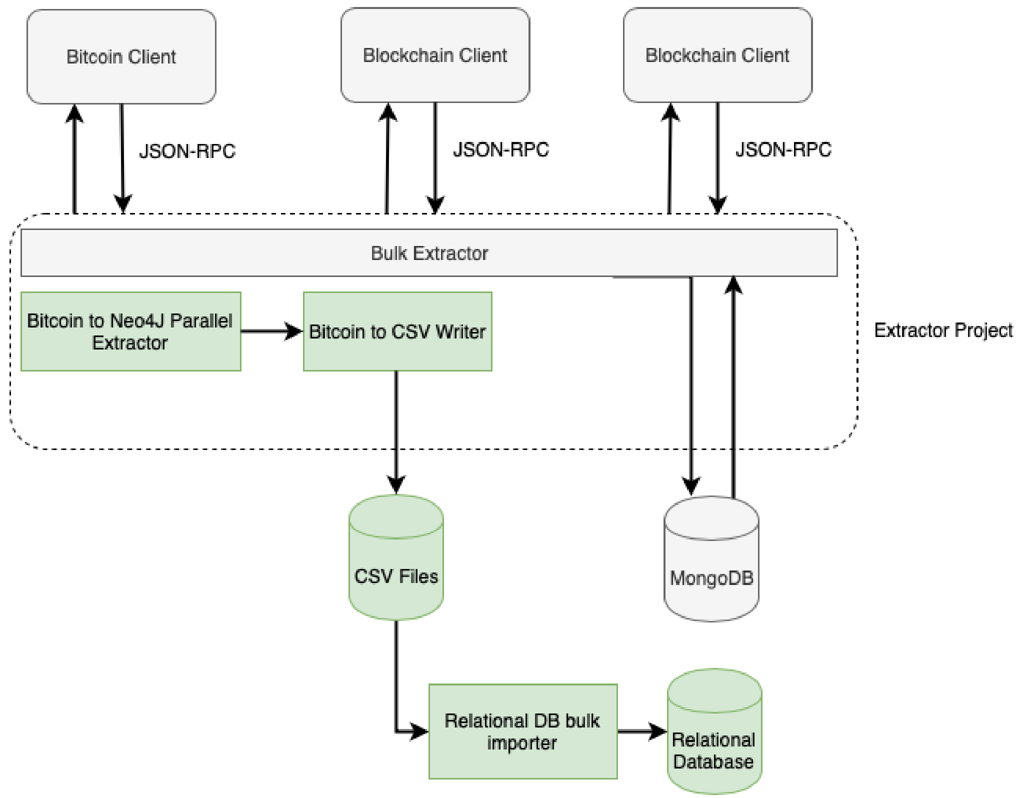

Figure 3.1: A architecture diagram displaying the new components (in green) we introduced to Max's Blockchain Health project while implementing Bitcoin to Neo4J historic data population

3.3 Challenges & Solutions:

3.3.1 Efficiency

With multiple cores at our disposal, it would only be logical to try and distribute the workload across the cores. Fortunately, WebFlux conveniently provides the functionality to do this; by adding `.parallel(n)` to the Flux stream, the workload is divided up into `n` rails (a rail is a subset of the work to do). Then by subsequently applying the `.runOn(Schedulers.parallel())` mapping, WebFlux is told to parallelise this work by running each rail on a separate core.

3.3.2 Job Failure Mitigation

A job is a unit of work that *Astrolabe* handles, such as a request to download a range of 100,000 Bitcoin blocks. When an error from the client is received, possibly due to being overwhelmed with requests or a temporary network issue, it would be a naive, inefficient solution to allow a single failure to cause the entire job to fail. Therefore we mitigated job failure by adding retry logic to RPC requests so that failures will first be handled by initiating a re-try mechanism with a delay (the delay necessary to allow time for recovery in the case of being overwhelmed). However, if an unexpected error occurs that we have not anticipated, the entire job may fail. Therefore, when performing the download, we ensured to do so in incremental batches. Specifically, we first download data for blocks 0-100,000 then 100,001-200,000 and so on. Therefore, if a failure does occur, progress from other blocks will have been saved from previously successful runs and only the batch which failed needs to be re-run.

3.3.3 Writing Concurrently from Several Threads

Although the above parallelisation improves the efficiency of the overall download process, it introduced the problem of multiple threads writing to a single CSV file concurrently; this led to data in the CSV files being malformed where threads have written data chunks in an interleaving fashion.

Fix Interleaving Writes with a Lock

Clearly, each line in the file needs to be written atomically, so we introduced a lock that each thread must hold in order to perform a write to a CSV file. However, we realised that allowing only one thread to write to the output file at any one time will create a significant bottleneck in the workflow; therefore, we sought an alternative solution.

Fix Interleaving Writes with Per-Thread Files

To enable parallel file writing and in order to circumvent the bottleneck of acquiring locks, we enabled each thread to write to a separate file. For example, the CSV file containing the block nodes will be named `block-data-thread-1.csv`, `block-data-thread-2.csv` and `block-data-thread-3.csv` when created by threads with ids 1, 2 and 3 respectively. With this solution, each write can occur without risk of another thread also attempting to write, so the necessity to acquire a lock for concurrent access no longer exists.

3.3.4 Duplicate Addresses

While generating the CSV files, we fetch the outputs of a transaction and the addresses each output is locked to. While generating the CSV, it wasn't possible to efficiently check if we had already seen an address before; instead, we had to write every address we see to the CSV file. Consequentially, when running the Neo4J bulk import job, we encountered an error as we attempt to define an address node twice. A solution considered was to use a flag `--ignore-duplicate-nodes` that can be used when invoking the bulk import tool. This would ignore any address nodes that have already been defined, solving our problem. However, as TokenAnalyst found in their work [27], the `--ignore-duplicate-nodes` flag massively slows down the import process, as Neo4J needs to check every single node to see if the one it is adding already exists. Their solution, which we adopted, was to use GNU's

`sort - u` command, which allows us to generate a new address CSV file with only the unique addresses. TokenAnalyst found this process to take around 30 minutes [27]; therefore, this was a valuable investment in order to speed up the overall import process.

3.4 Result

All Bitcoin Blockchain data was able to be downloaded, parsed and written as CSV files across a 12 hour period up to block 570,000. Due to the process of human intervention in our approach, necessary to mitigate the effects of a job failing as described above, the download was not automatically continuous, such that there were idle times between jobs finishing and the next being started. Additionally, manual intervention was required post-download in order to augment the Bitcoin data with price information and entity relationships, as described in section 4 and section 5 respectively.

Chapter 4

Fetching Historical Price Data: *Compass*

Users of the tool should be able to view the value of transactions at the time they took place, in several fiat currencies (such as GBP, USD and EUR). To achieve this functionality, the historical price index for Bitcoin must be collected and stored for the entire time spanning from the infancy of Bitcoin to the present day. Therefore, the objective of this contribution is to collect the historical exchange rates for every block from height 0 to 570,000, and to then store the various exchange rates as metadata with each block.

4.1 Source of Price Data

Fortunately, *CoinDesk* has an API providing the historical Bitcoin price index data from late July 2010 onwards [30]. The API can provide the daily price index for each day for a provided date range, supporting several fiat currencies.

4.2 Storing the Price Data

Since the historical price data retrieved from *CoinDesk* is at the granularity of one data point per day, and a block in Bitcoin is mined approximately every 10 minutes, it would be more appropriate for the price data to be stored with each block in the database. Storing the data with each transaction output would not provide any additional value, and would lead us to encounter a greater storage overhead and computational effort in associating the price data with each Bitcoin transaction output. Since each output is associated with the transaction which produced it, and each transaction associated with the block it was mined in, we can easily find the price index data for an output through a 2 hop graph traversal.

4.3 Matching Price Data to Bitcoin Data

As discussed in the previous section, it would be appropriate to store the historical exchange rate data with Bitcoin's block data. Using Python3, we created a program to:

1. Accept CSV file names (for the block CSV files) as input (or a regex for matching files)

2. Read through each line of the CSV (representing a single block)

3. Write a new output line with each row (block) augmented with the price data at the time the block was mined in GBP, USD and EUR.

Since at the time of writing, the current Bitcoin block height was $>$ 570,000, it would not be efficient to perform a price index request to *CoinDesk* for each block, repeated each fiat currency we support. Therefore, *Compass* first checks a local cache to see if the price data is already available for a particular date. If not, it performs a fetch which collects the data for that date and the following 300 days, populating the cache with the additional data. Requesting 300 days in advance of data reduces the total number of requests made, and the overhead associated with making each request. We do not request data at the maximum range (from Bitcoin's infancy to present day) in order to strike a balance between reducing the number of requests and keeping response sizes reasonable.

4.4 Using the Price Data

The price data is now associated with each Bitcoin Block in CSV format. These files can be used in the database import process in section 6 and the exchange rate figures can be stored with each block entry. When an exchange rate for a particular transaction is required, its relationship with the block it was mined in can be traversed and the historical exchange rate for the time the transaction occurred can be found.

Chapter 5

Entity Tagging: *Quadrant*

As discussed in background section 2.4.1, it is possible to associate public addresses with known entities, such as exchanges, pools and services. *Wallet Explorer* hosts a wealth of data mapping entities (e.g. exchanges, pools, services etc.) to the public addresses they are known to operate under [31]. Therefore, this objective is to collect address tagging information from *Wallet Explorer* and to associate it with the historical Bitcoin Blockchain data we have previously collected.

5.1 Retrieving Wallet Data

Wallet Explorer does not provide functionality for users to download the data they serve. The data can only be viewed by navigating to each wallet, then to their addresses, then through a series of paginated tables providing the addresses for that entity.

Our solution to this problem was to build a web scraping program, *Quadrant*, which navigates the site and builds a mapping for each wallet (entity) to all of its addresses, which can then be written to some output file for mapping against the stored bitcoin data at a later stage.

5.1.1 Building the Scraper

We used a popular open-source web-scraping framework *scrapy* for implementing *Quadrant*. We generate a new web crawler by extending the framework's 'CrawlSpider' class. In this class, we define the bounds of the crawl and define methods to parse and extract the data for each page it visits. For example, we created rules to allow the crawler to navigate to links with 'addresses' in the URL and then defined how to extract the wallet name, each of the addresses from the table and how to follow links to paginate the table in order to view all addresses for that wallet.

However, when we initially ran *Quadrant*, we continuously received 403 (Forbidden) error responses for the majority of requests. After investigation, we discovered the likely cause of this was due to anti-scraping measures many websites employ. In order to circumvent these anti-scraping measures, we experimented with several approaches recommended by *scrapy's* documentation titled 'Avoid getting banned' [1]; these included rotating user agents, disabling cookies, use download delays or use a pool of rotating IP's.

[1] https://docs.scrapy.org/en/latest/topics/practices.html?#avoiding-getting-banned

Implementing Delays

This approach worked; we were able to visit many pages without being served a 403 response as before; however, this took a very long time. A delay of more than 2 seconds was required to circumvent the website's scraping detection such that the delay solution works; therefore, this would be an inefficient solution for scraping thousands of pages containing the required data on the *Wallet Explorer* site.

Using Rotating Proxies

We integrated into *Quadrant scrapoxy* which allows us to hide the scraper behind a cloud provider (in this instance, AWS). *Scrapoxy* creates a pool of proxies and routes all requests through the pool of proxies. We configured *scrapoxy* to use a personal Amazon EC2 account and was able to scale up to 20 EC2 instances when performing the scraping (to enable greater concurrency, but limited at 20 because the account is restricted to 20 instances at the free tier). Through trial and error, we were able to configure the number of concurrent requests per proxy to 5, which further maximises concurrency without encountering error responses due to anti-scraping measures.

5.2 Results

Using 20 EC2 instances, the scrape completed on 14th April 2019 at 10:07:42 successfully after 1 day, 19 hours and 12 minutes, in which time *Quadrant* received 240,411 HTTP responses. This amounted to generating a 920MB data file containing entity to address mappings. Due to the concurrency of the retrieval process, distinct threads may have fetched different pages of data for each wallet. To simplify the scraping tool and so that it runs as fast as possible, the thread will write that wallets information as a new entry in the JSON. However, this will lead to multiple duplicate keys, such as in the example below.

```
[
{"wallet": "wallet1", "addresses": ["add1", "add2"]},
{"wallet": "wallet2", "addresses": ["add3"]},
{"wallet": "wallet1", "addresses": ["add4", "add5"]}
]
```

However, we would like the desired format to be of the format:

```
{
"wallet1": ["add1", "add2", "add4", "add5],
"wallet2": ["add3"]
}
```

Thankfully the size of the file to be processed was still small enough to load into memory, enabling the implementation of performing this transformation to be simpler. We created a simple script which iterated through the input file and generated a new dictionary containing wallet names as unique keys and concatenated lists of addresses from each entry matching the wallet name. We then wrote this out in JSON format to a new file.

5.3 Performing the Address Matching

From step 5.1 we have a JSON file mapping each entity type to a list of addresses. After importing the Bitcoin Blockchain into CSV format, and de-duplicating addresses, we have a CSV containing all distinct addresses to have ever been used in a Bitcoin transaction. For each address that is controlled by an entity, we now need to generate a new `HAS_ENTITY` relationship and write it in CSV format. If we were to take a naive approach here, where we compare all Bitcoin addresses against each address we have a mapping for, we would have an algorithm which performs $\mathcal{O}(nm)$ address matches, where n is the total number of Bitcoin addresses and m the number of addresses we have an entity mapping for.

We took a more efficient approach and used a Trie data structure. The Trie data more efficiently stores Bitcoin addresses data since it only needs to store once the common prefixes of the millions of addresses. The primary benefit, though, is for performing the address matching. For each Bitcoin address, we only need to walk, at most, the height of the tree, so we only require $\mathcal{O}(n)$ address matches. We were able to implement this approach using Google's pygtrie library.

Chapter 6

Database Population

We now use the data retrieved for all historical Bitcoin transactions in section 3 and the augmented datasets developed in sections 4 and 5 and import them into a Neo4J database using the Neo4J Bulk Import tool.

6.1 Why Neo4J?

Graph DB's provide efficient traversing of nodes; allowing million of connections to be traversed per second per core [32]. Scaling independently to the size of the dataset, a graph database is excellently suited for storing the vast, complex dataset constituting the Bitcoin Blockchain. Neo4J is an open-source, NoSQL graph-database providing ACID compliant transactions.

6.1.1 Other DB Solutions

It was very clear from the outset that a graph database would be required; it fitted the problem of needing to traverse many relationships quickly. Since we are working with such a large dataset, we must preempt the performance issues associated with searching for data in a large database. In graph databases, performance remains almost constant as the size of the dataset grows.

Flexibility was also an important consideration; it would be necessary to easily adapt the data model over time, such as when introducing a new clustering heuristic or adding a new property to a node; graph databases provide flexibility to adapt the data structure far easier than in traditional relational databases. Additionally, since the investigation tool we were building would display data in a graphical format, it made sense to reflect this in the format the data is stored.

Other (free) graph DB technologies include Orient DB, Graph Engine, GraphDB Lite, Titan, and MapGraph. Neo4J was selected due to our familiarity with the technology, its large support community and due to the graph persisting data to disk, unlike some other technologies that are in-memory-based solutions which would not be possible due to the size of our dataset.

6.1.2 Bulk Import Tool

Neo4J provides functionality for bulk importing data [33]. A Neo4J blog describes an example use of this functionality: importing a vast 66GB dataset from Stackoverflow into a new Neo4J database [34]; however, this is less than a third the size of the Bitcoin blockchain (approximately 197GB at the beginning of January 2019 [35]).

The import tool is designed to take advantage of the hardware at its disposal; ideal for our use-case where we have a large amount of SSD and processing power available, and we need to ensure we can take full advantage of it.

6.2 Database Design

The graph Database will consist of 'nodes' which will store each data entry and relationships which exist between nodes, which describe the relations between the data.

6.2.1 Data Nodes

- `BLOCK`: A Bitcoin block, unique id being the block hash.
- `TRANSACTION`: A Bitcoin transaction, unique id the txid property.
- `OUTPUT`: A transaction output, unique id the txid property concatenated with the outputs index in the transaction.
- `COINBASE`: The special type of input that mints new bitcoin - has not been produced by a transaction.
- `ADDRESS`: A Bitcoin public address, unique id the public address itself.
- `ENTITY`: A known entity as collected from walletexplorer.com in section 5.

6.2.2 Relationships

We created several types of relationships to exist between nodes in Neo4J. The relationships can be seen in a visual representation below in figure 6.1.

- `CHAINED_FROM`: Exists between two blocks; represents the relationship between a block and it's parent block.
- `MINED_IN`: Exists between a transaction and a block. Represents the relationship between a transaction and the block it was mined in.
- `LOCKED_TO` Exists between a transaction output and an address. Represents the relationship between an output, and who it can be spent by.
- `INPUTS` Exists between a transaction output and a transaction. Describes the relationship between an output and the transaction it later funds.
- `OUTPUTS` Exists between a transaction and a transaction output. Shows the relationship between a transaction and the new outputs it generates.

- `COINBASE` Exists between a coinbase node and a block. Represents the special type of input to a transaction which generates new bitcoin, and is associated with a block as it was the miners' reward for successfully mining the block.
- `HAS_ENTITY` Exists between entities and the addresses they are known to control (see fig 6.2).

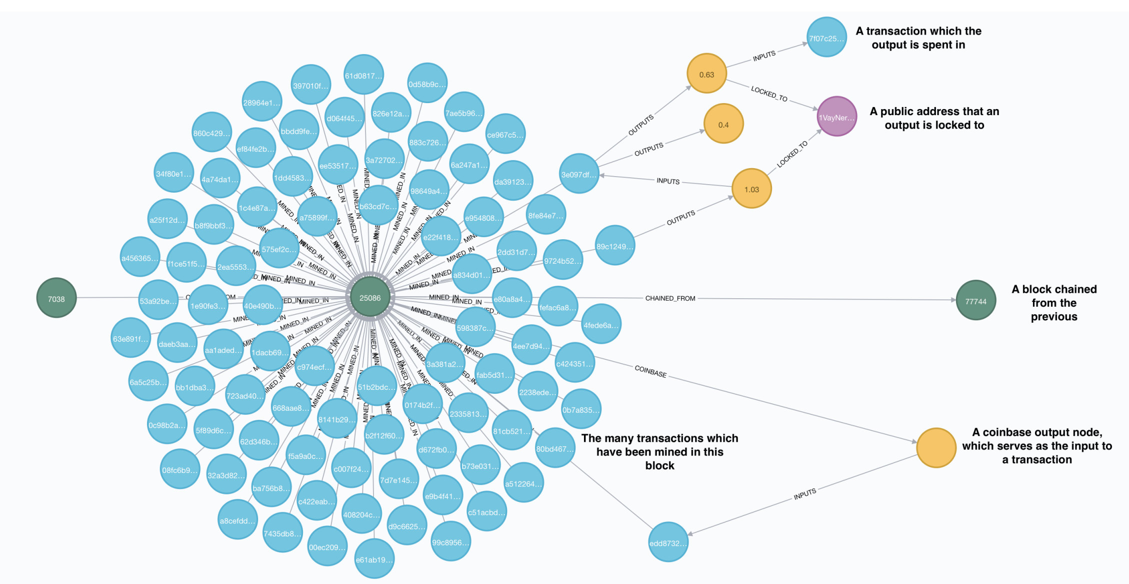

Figure 6.1: The nodes and relationships as visualised using the Neo4J Web UI

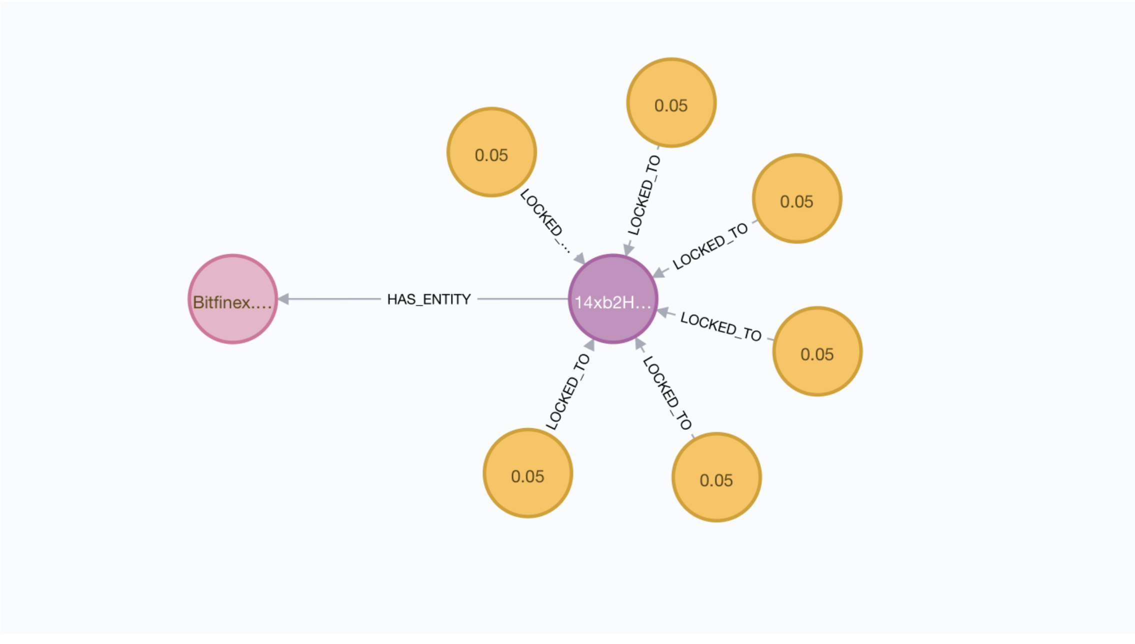

Figure 6.2: The new `HAS_ENTITY` relationship visualised between a new Entity node and a bitcoin address.

6.3 Invoking the Import Job

We created a script to wrap the execution of the import job, providing the ability to version control the script and to ensure reproducibility of the import process. This script first uses *globbing* to build the list of input files for each type of node and relationship. It then invokes the import command, as shown below. Some additional arguments were `--max-memory`, to override the default 90% memory usage constraint in an effort to alleviate memory issues. The argument `--ignore-missing-nodes` was to prevent a long job failing simply due to a single relationship referring to a non-existing node, and to instead rely on these bad relationships being reported to the file as defined by the `--report-file` option for manual investigation.

```
$1/bin/neo4j-admin import
  --nodes:ADDRESS $address_files_all
  --nodes:BLOCK $block_files_all
  --nodes:COINBASE $coinbase_files_all
  --node:OUTPUT $output_files_all
  --nodes:TRANSACTION $transaction_files_all
  --nodes:ENTITY $entity_files_all
  --relationships:CHAINED_FROM $relation_chained_from_files
  --relationships:COINBASE $relation_coinbase_files
  --relationships:INPUTS $relation_inputs_files
  --relationships:LOCKED_TO $relation_locked_to_files
  --relationships:MINED_IN $relation_mined_in_files
  --relationships:OUTPUTS $relation_outputs_files
  --max-memory 95%
  --ignore-missing-nodes true
  --report-file "neo4j-import-debug-report.log"
```

6.4 Challenges & Solutions

6.4.1 Memory Issues

While attempting to perform the bulk import, we encountered an issue where the import job would fail with no output other than the message that the process was killed. This occurred on repeated runs at around 28% progress into the graph node creation process. After investigation, we found that this issue was being caused by Neo4J running out of available memory; we rectified this by allocating a much larger amount of Swap memory for the VM. The total amount of memory available (including swap) increased to over 30GB. This allowed the import tool to progress using swap once memory had been exhausted, albeit with increased memory access latency.

6.4.2 Query Latency Issues

Upon successful population of the database, we performed simple entity lookups to begin to verify the correctness and performance of the database. However, we discovered immediately that simple searches for an address node using the tool takes an unacceptably long time (10

minutes 45 seconds to find an address node by its address and find its immediate neighbours). We used indexing to solve this issue.

6.4.3 Creating indexes

Indexes are useful for finding the starting point of graph traversal. *Radar* requires exactly this functionality to support features to find a particular Bitcoin address, used as the starting point of an investigation before expanding out neighbours to trace the flow of funds. The initial search for an address node is where the latency currently exists; therefore, the first index we created was on the `address` property of the `ADDRESS` node. We create this index by executing the command `CREATE INDEX ON :ADDRESS(address)` and using the command `CALL db.indexes` to track the progress of its population.

Once the index was created, the result for an address node search returned almost immediately. However, when trying to find the relations of the address neighbours (including non-address nodes), we encountered the same performance issues. Therefore, we created indexes for each node type using the unique ID that each node would be fetched with. The complete set of indexes created are:

- `:ADDRESS(address)`
- `:BLOCK(hash)`
- `:ENTITY(name)`
- `:OUTPUT(outputId)`
- `:TRANSACTION(transactionId)`

6.5 Import Result

The import operation for the first 570,000 Bitcoin Blocks completed in 5 hours, 32 minutes and 555 seconds. The operation consumed 22.96GB of memory and created 1.964 billion nodes, 3.52 billion relationships and 3.03 billion properties. There were no errors or bad relationships reported.

Chapter 7

Backend API: *Loran*

In order to provide an interface for interacting with Neo4J, and in order to support more complex functionality such as clustering and path finding, we developed *Loran* to support the features of the *Radar* as described in section 9.

7.1 Technology Choices

Loran was developed in Java using Spring Data as the predominant library used for communicating with Neo4J.

Spring Data provides powerful repository and object-mapping abstractions that can be used to define a data model and interact with the underlying data store through abstractions with relative ease. Spring Data has specific support for Neo4J, utilised by creating a repository that extends their `Neo4JRepository<>` interface. Spring Data repositories provide 'out-of-the-box' database interaction functionality; for example, queries can be derived from the repository method names that are defined. Spring Data also has support for many other underlying data repositories, such as MongoDB, Redis or Apache Cassandra. Such an abstraction would allow for the flexibility of changing the core underlying data store, or for the introduction of new data sources from different types of data stores.

7.1.1 Alternative Technologies

An alternative technology choice, which is often a popular approach, would be to create *Loran* using Python Flask. Python Flask was considered, and was an almost equal candidate for technology choice, but we ultimately decided to choose Java & Spring Data due to the huge community that exists for Spring MVC and its excellent documentation. Additionally, we were aware of the future potential of integrating Max Baylis Health Monitoring project [see 2.8.2] (written in Java) with the interface to Neo4J, in which case implementing *Loran* in Java using Spring Data would make such an integration far easier.

7.2 API Design

Loran follows a REST design pattern; generally, IDs are used to retrieve (HTTP GET request) an entity by passing it as a path variable. Any further options, such as data required for filtering, exist as an optional query parameter.

Listing 7.1: Get an address using the unique full address. Several optional query parameters for filtering by time, price and enabling clustering and node limiting.

```
GET /bitcoin/address/{address | string}
    ?startTime={time filter start since epoch | string}
    &endTime={time filter end since epoch | string}
    &startPrice={price filter start | double, as string}
    &endPrice={price filter end | double, as string}
    &priceUnit={currency units of price filter | string of 'btc', 'gbp',
        'usd' or 'eur'}
    &inputClustering={true/false | boolean}
    &nodeLimit={node limit number | integer}
```

Listing 7.2: Get an entity using the unique name of the entity . All query parameters are optional for filtering.

```
GET /bitcoin/entity/{entity name | string}
    ?startTime={time filter start since epoch | string}
    &endTime={time filter end since epoch | string}
    &startPrice={price filter start | double, as string}
    &endPrice={price filter end | double, as string}
    &priceUnit={currency units of price filter | string of 'btc', 'gbp',
        'usd' or 'eur'}
    &nodeLimit={node limit number | integer}
```

Listing 7.3: Get an output with a unique output ID. All query parameters are optional for filtering.

```
GET /bitcoin/output/{output id | string}
    ?startTime={time filter start since epoch | string}
    &endTime={time filter end since epoch | string}
```

Listing 7.4: Get a transaction with a unique transaction ID (txid). All query parameters are optional for filtering.

```
GET /bitcoin/transaction/{transaction id | string}
    ?startTime={time filter start since epoch | string}
    &endTime={time filter end since epoch | string}
    &startPrice={price filter start | double, as string}
    &endPrice={price filter end | double, as string}
    &priceUnit={currency units of price filter | string of 'btc', 'gbp',
        'usd' or 'eur'}
    &nodeLimit={node limit number | integer}
```

Listing 7.5: Get a block with a unique block hash

```
GET /bitcoin/block/{block hash}
```

Listing 7.6: Find a path between two addresses using their full address strings

```
GET /bitcoin/shortestPath/{start address}/{end address}
```

7.2.1 Responses

A typical response will contain all information for the requested entity, including its immediate neighbours. For example, the request
`GET /bitcoin/output/fbec1...`[1] will produce the response shown in listing 7.7. As shown, searching for an output returns the ids of the neighbouring nodes (see neighbouring node `lockedToAddress`) which can subsequently be used in API calls that fetch neighbouring node data and, therefore, enable a user to trace the flow of funds.

Listing 7.7: GET Output Example Response


```
{
    "outputId": "
        fbec1c21ca91d4e5baf55f305b05d294755b4fd069d149344b2104b708e42873
        -0",
    "value": 50,
    "producedByTransaction": {
        "transaction": {...}
        "eurValue": 0,
        "usdValue": 0,
        "gbpValue": 0,
        "timestamp": 1232709019
    },
    "inputsTransaction": {
        "transaction": {...},
        "eurValue": 0,
        "usdValue": 0,
        "gbpValue": 0,
        "timestamp": 1233004161
    },
    "lockedToAddress": {
        "address": {
            "address": "1BENJudbbZ8dfTwFtCLuJNWMTtBLE2bZa",
            "entity": null,
            "hasLinkedAddresses": true
        }
    }
}
```


The response for the path finder method takes on a different format. For example, executing the path finding query on addresses *1CrnUia9wfeNFbdwKJNj89YqA6qetvYTTE* and `17c6L9JUGVenn6CfqXuB93L3Tk8Tbzefui` returns the response shown in listing 7.8.

The start and end nodes are fully defined, and each intermediate node on the path (each potentially of a different type) are included in the `intermediateNodes` array, with the links that connect each of the nodes included in the `rels` array.

[1] fbec1c21ca91d4e5baf55f305b05d294755b4fd069d149344b2104b708e42873-0

Listing 7.8: Response to a path find query


```
[
    {
        "startNode": {
            "address": "1CrnUia9wfeNFbdwKJNj89YqA6qetvYTTE",
            "outputs": [
                {...}
            ],
            "entity": null,
            "inputHeuristicLinkedAddresses": null,
            "hasLinkedAddresses": false
        },
        "intermediateNodes": [
            {...},
            {...},
            {...}
        ],
        "rels": [
            {...},
            {...},
            {...}
        ],
        "endNode": {
            "address": "17c6L9JUGVenn6CfqXuB93L3Tk8Tbzefui",
            "outputs": [
                {...}
            ],
            "entity": null,
            "inputHeuristicLinkedAddresses": null,
            "hasLinkedAddresses": false
        }
    }
]
```


7.3 Implementation

7.3.1 Overall Design

The API is defined in a class named 'BitcoinController '. Defining an API using this class is achieved using Spring Data by annotating the class with the `@RestController` and `@RequestMapping` annotations. Requests are handled by each of the methods shown in the BitcoinController class in figure 7.1. A request is sent to the service which coordinates the data to be returned using a number of the repositories. There exists one repository for each type of entities.

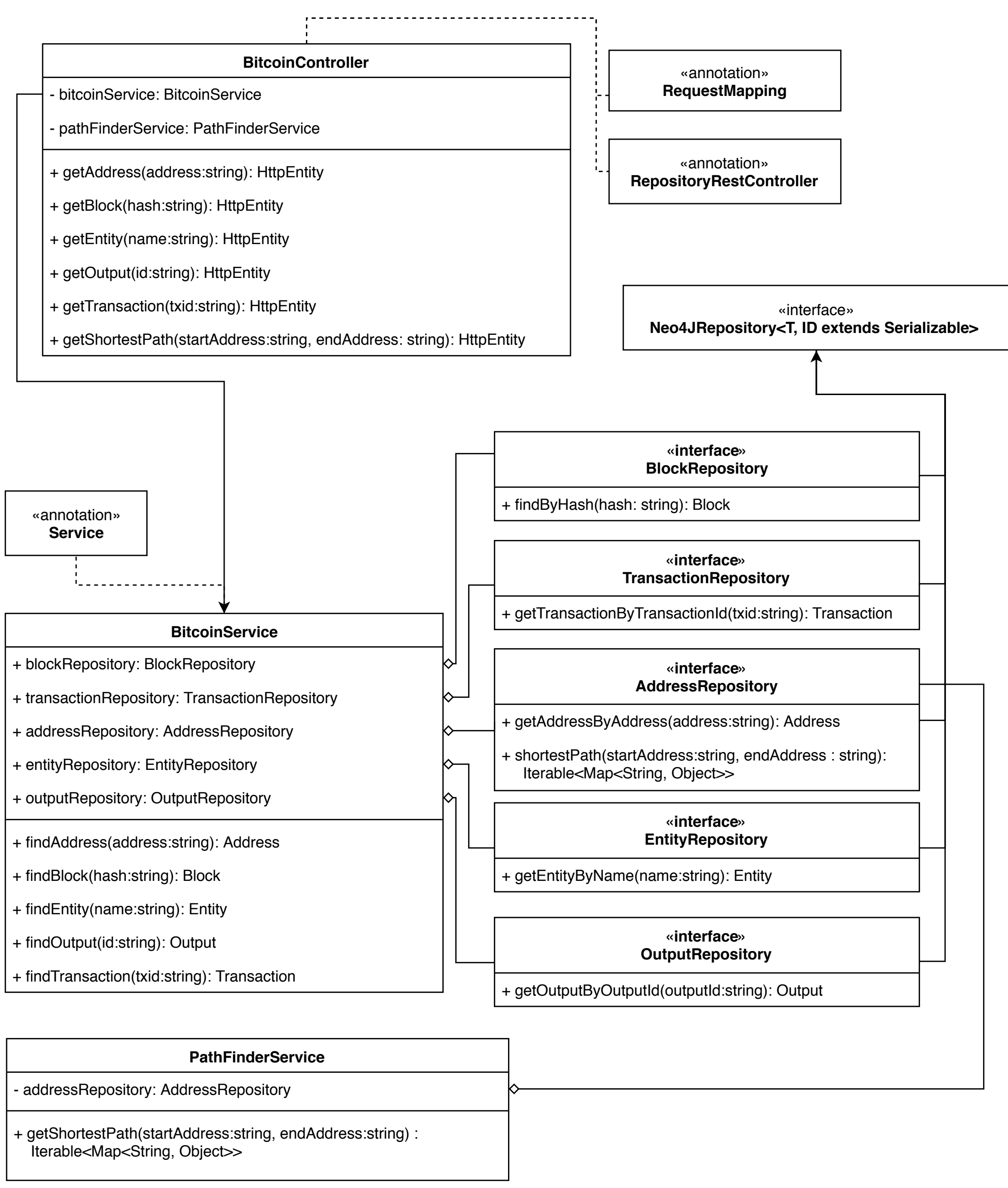

Figure 7.1: *Loran* Implementation UML Diagram

7.3.2 Node Entities

A data model can be defined by creating classes and annotating them using the Spring Data `@NodeEntity` annotation; Relationships from the database are represented using the `@Relationship` annotation. An example of a node entity class implementation is the transaction node type shown in listing 7.9.

The annotation `@NodeEntity` is used to tell Spring Data this class represents the data model of a node in Neo4J. The `@Id` annotation identifies the `id` field as the node's unique graph identifier from Neo4J. `@Relationship` annotations on the class change the Cypher query sent when fetching a node, in order to additionally fetch neighbouring node data by traversing the relationships defined. This helps produce responses as shown in listing 7.7 where the neighbouring node data is populated.

7.3.3 Serialising Node Entities

The annotation `@JsonIgnoreProperties` is used to prevent infinite recursion when serialising responses to be returned by *Loran*. For instance, if a transaction references an output node that it produces, and each output references the transaction node that produces it, the serialisation would recurse infinitely until a stack overflow error is encountered, at which point serialisation fails. Additionally, methods omitted from listing 7.9 for brevity are the 'Getter and Setter' methods that exist for the properties; these methods help define the fields that should be serialised. For instance, no Getter exists for the `id` field since it is not required in the response.

Listing 7.9: A transaction node entity

```
@NodeEntity(label = "TRANSACTION")
public class Transaction {
    @Id
    @GeneratedValue
    private Long id;

    private String transactionId;

    @Relationship(type = "MINED_IN", direction = Relationship.OUTGOING)
    private Block minedInBlock;

    @JsonIgnoreProperties("transaction")
    @Relationship(type = "INPUTS", direction = Relationship.INCOMING)
    private List<InputRelation> inputs;

    @JsonIgnoreProperties("inputsTransaction")
    @Relationship(type = "INPUTS", direction = Relationship.INCOMING)
    private Coinbase coinbaseInput;

    @JsonIgnoreProperties("transaction")
    @Relationship(type = "OUTPUTS")
    private List<OutputRelation> outputs;

}
```

7.3.4 Implementing Repositories

As shown in the UML diagram 7.1, repositories all exist as interfaces which extend the `Neo4JRepository` interface. In doing so, many of the queries are automatically inferred based on the names of the methods defined on the interface. For example, the implementation of the AddressRepository interface is shown in listing 7.10. The method `getAddressByAddress` requires no query definition since Spring Data infers the query by the name of the method, and knows the type as the interface provides the `Address` type as one of the generic type arguments when extending Neo4jRepository.

Comparatively, the method shortestPath does require a query definition using the Spring Data `@Query` annotation, as shown in listing 7.10; this is as the query is more specialised and cannot be inferred.

Listing 7.10: AddressRepository interface definition

```
public interface AddressRepository extends Neo4jRepository<Address, Long>
    {

    Address getAddressByAddress(@Param("address") String address);

    @Query("MATCH" +
            "(a1:ADDRESS{address:{0}})," +
            "(a2:ADDRESS{address:{1}})," +
            "p = shortestPath((a1)-[:INPUTS|:OUTPUTS|:LOCKED_TO*..100]-(
               a2))" +
            "RETURN a1 as startNode, nodes(p) as intermediateNodes,
               relationships(p) as rels" +
            ", a2 as endNode")
    Iterable<Map<String, Object>> shortestPath(String sourceAddress,
        String destinationAddress);
}
```

7.3.5 Implementing Path Finding

The implementation of the path finding functionality can be seen in listing 7.10. Neo4J provides the method `shortestPath` that can be used to find a path between two nodes. We restricted the type of paths to be of type `INPUTS`, `OUTPUTS` or `LOCKED_TO` in order for the path to represent the flow of funds, rather than paths existing simply due to all nodes on the blockchain, by definition, are connected via their relationship with the blocks they are mined in, and blocks with each other.

Chapter 8

Clustering: *Balestilha*

The *multi-input* clustering heuristic, as described in section 2.3.1 and in the paper by Sarah Meiklejohn et al. is one the most effective clustering heuristics for Bitcoin [18]. In this section, we present the several implementations of clustering algorithms, which uses the *multi-input* heuristic. We attempted clustering with several approaches due to issues discovered while running each clustering algorithm. We also present the successful implementation in section 8.5.

8.1 Algorithm

The rough outline of the algorithm is shown below.

- for each transaction tx on the Blockchain
  - Fetches inputs ins of tx
  - Gets all addresses as where each a in as is an address which an input in from ins is locked to
  - Clusters the addresses as

8.2 Java & Spring Data Approach

The first approach taken was implemented as an extension of the Neo4J API

- Created a new clustering service in *Loran*
- Fetched all transactions from Neo4J
- For each transaction, make additional requests to Neo4J for inputs and addresses the inputs are locked to
- When a transaction has multiple inputs, add all addresses that input it to a set in memory
- Execute a query to create a new relationship between every address in the set (such that it creates a totally connected sub-graph containing the addresses)

Executing this algorithm for a local instance of the database, which contained a subset of the Bitcoin Blockchain data (blocks 0-2000 only). The algorithm successfully completed and screenshots of the resulting database, with the new relationships introduced, can be seen in figure 8.1 and 8.2.

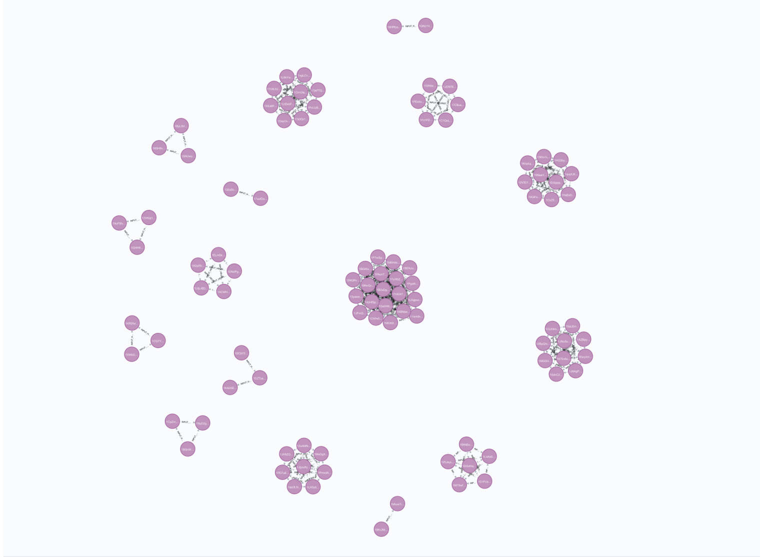

Figure 8.1: Neo4J browser view of the several clusterings of addresses which all feed the same transaction.

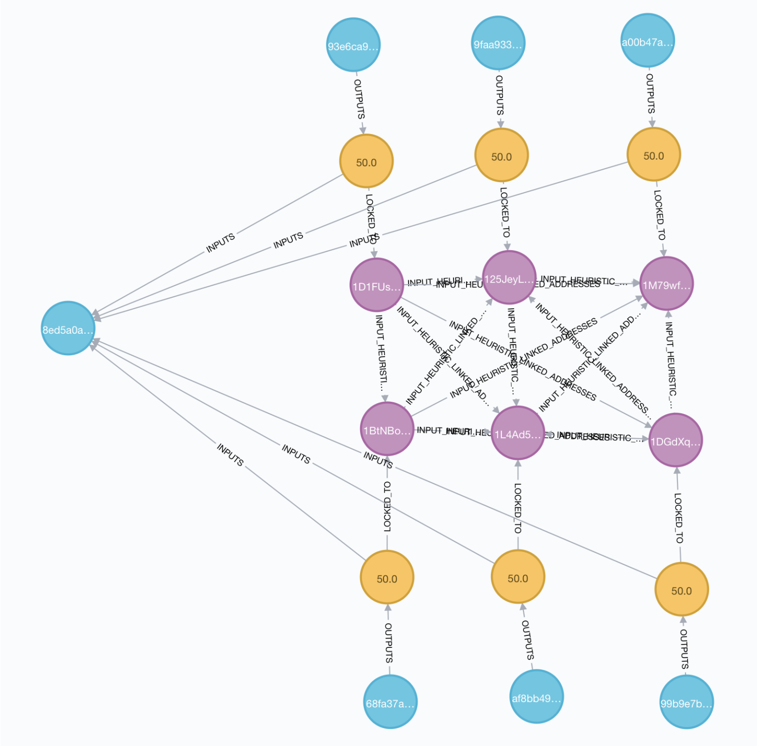

Figure 8.2: Neo4J browser view one one of the clusterings of addresses with their neighbours expanded to show they all have outputs locked to them which feed the same transaction

8.2.1 Challenges

An issue we anticipated would occur was an issue due to to the size of the dataset we are working with; we encountered a memory overflow error while the algorithm was attempting to load Bitcoin transactions into memory.

We solved this issue using paging. Paging allows us to break our request for all transactions to several requests for batches of transactions. Fetching transactions in batches allowed each batch's transactions to be processed before fetching the next batch. This helped tackle memory constraints as we didn't need to load **all** Bitcoin transactions up front. See code in Listing 8.1

Although progress was now being made more steadily using paging, we still experienced performance issues while trying to cluster all Bitcoin addresses. After running the above algorithm for 24 hours, only 150,000 transactions had been processed. This was infeasibly slow, most likely due to the overhead of using Spring Data to create new relationships in Neo4J between all addresses in each address cluster; it was clear a more efficient implementation was required.

Listing 8.1: Java Implementation using Paging

```
public ResponseEntity clusterByInput() {
    //Pageable item: start at page 0, fetch 50 items
    Pageable pageable = PageRequest.of(0, 50);

    while (true) {
        //Fetches transactions for this page
        Page<Transaction> allTransactions = transactionRepository.findAll
            (pageable);
        allTransactions.forEach(transaction -> {

            if (transaction.getInputs() == null || transaction.getInputs
                ().size() < 2) {
                //coinbase input or only one input
                return;// just skips this iteration only
            }
            //Fetch inputs for this transaction
            List<InputRelation> transactionInputs = transaction.getInputs
                ();
            Set<Address> addressesSpendingTransactionInputs = new HashSet
                <>();

            transactionInputs.forEach(inputRelation -> {
                String inputId = inputRelation.getInput().getOutputId();

                //Fetches the entire Output node from the database, so it
                    's
                //relation fields are populated
                Output refetchedTransactionInput = getOutputById(inputId)
                    ;
```

```
                Address addressSpendingTransactionInput =
                    refetchedTransactionInput.getLockedToAddress();

                //add every distinct address to a set
                if (addressSpendingTransactionInput != null) {
                    addressesSpendingTransactionInputs.add(
                        addressSpendingTransactionInput);
                }
            });

            //iterate through the set, link every item in set with evrey
                other
            addressesSpendingTransactionInputs.forEach(address -> {
                address.setInputHeuristicLinkedAddresses(
                    addressesSpendingTransactionInputs);

                // Saves the updated addresses back to Neo4J repo at
                    depth 0
                this.addressRepository.save(address, 0);
            });

            System.out.println("completed for tx" + transaction.
                getTransactionId());

        });

        if (!allTransactions.hasNext()) {
            //reached the last page: terminate
            break;
        }

        //fetches the next page to request
        pageable = allTransactions.nextPageable();
    }

    return ResponseEntity.status(200).body("Clustering Complete");
}
```

8.3 Cypher Query

The intention of experimenting with a pure Cypher implementation was if we could fully specify the algorithm in Cypher, then Neo4J could potentially parallelise/optimise the workload more efficiently if it knows the entire algorithm up-front. We also hoped that using Cypher would improve performance by bypassing the necessity to interact with the database using Spring Data, which will be adding some overhead.

We were able to translate the above algorithm in Listing 8.1 to a single Cypher query. The Cypher implementation is shown in Listing 8.2. The query matches all transactions that have more than one input, then finds the addresses each input is locked to. It uses the `UNWIND` commands to generate all pairs of addresses in the address list for a particular transaction. The clause `WHERE id(first) < id(second)` ensures a pair of addresses only occurs once;

rather than two pairs existing for {a,b} and {b,a} where a and b are both addresses.

Listing 8.2: Cypher Implementation

```
MATCH (t:TRANSACTION)
WHERE size((t)<-[:INPUTS]-()) > 1
WITH [(t)<-[:INPUTS]-(:OUTPUT)-[:LOCKED_TO]->(a:ADDRESS) | a] as
    addresses
UNWIND addresses as first
UNWIND addresses as second
WITH addresses, first, second
WHERE id(first) < id(second)
MERGE (first)-[:INPUTS_SAME_TX]-(second)
```

Before we could execute the query, we first had to clean up the mutations made to the database by running earlier clustering implementations. To do this, we created a Query to delete all instances of the relationships we added during the first attempt.

```
MATCH (:ADDRESS)-[r:INPUT_HEURISTIC_LINKED_ADDRESSES]-(:ADDRESS)
DELETE r
```

8.3.1 Challenges

When executing the query shown in Listing 8.2 we encountered performance issues where several hours would elapse with no apparent progress being made.

We investigated this issue using the Cypher `PROFILE` command to inspect the performance of this query. The performance report identified issues in the earlier stages where the query would require the search of all nodes across the database when matching transactions, inputs and addresses. Since we want to match all nodes, indexes cannot be leveraged to retrieve nodes, so we loose the performance benefits of indexes and therefore introduce a significant bottleneck in the query. Furthermore, investigating the issue using the `htop` command on the *Satoshi* VM, we could see that core utilisation was extremely low (almost completely idle).

The main difficulty with this approach is that executing a very expensive query becomes almost like a 'black-box' in that it is difficult to understand what Neo4J is doing, since we are using the Community edition of Neo4J, so we do not have access to logging options that can be used to help us.

8.4 Clustering on demand

This approach has the intention of tackling the performance issues by only performing clustering when and where it is required, rather than attempting to perform clustering for the entire Bitcoin Blockchain in advance.

By knowing exactly which address we want to perform the clustering for, and leveraging the performance enhancements provided by indexes on address, transactions and outputs when

fetching them using their respective ID's, we were able to craft an algorithm to implement clustering on demand.

This implementation heavily relied on Java 8 streams, specifically using parallel streams to introduce concurrency in the clustering process, increasing the efficiency of this operation; critical for an on-demand implementation.

The implementation of this algorithm led me to discover that the initial implementation (Listing 8.1) was not correct. This heuristic is transitive, such that if addresses A and B input the same transaction, and B and C input the same transaction then A, B and C can all be considered as under the control of the same user. The implementation in Listing 8.1 does not take this transitivity into account. However, we ensured transitivity was encountered before in the on-demand clustering algorithm in Listing 8.3.

Listing 8.3: Java Implementation of on demand clustering

```
private void performInputClustering(Address addressNode, Date start, Date
     end) {
    Set<Address> linkedAddresses = transitiveInputClustering(addressNode,
        new HashSet<>(), start, end);
    addressNode.setInputHeuristicLinkedAddresses(linkedAddresses);
}

private Set<Address> transitiveInputClustering(Address addressNode, Set<
    Transaction> exploredTransactions, Date start, Date end) {
    //a stream of transactions which all have inputs locked to this
        address
    Stream<Transaction> allTransactionsThisAddressInputs =
        getTransactionsForAddress(addressNode, start, end);
    HashSet<Transaction> thisAddressesTransactions =
        allTransactionsThisAddressInputs.collect(Collectors.toCollection(
        HashSet::new));

    //removes all transactions we've already seen
    thisAddressesTransactions.removeAll(exploredTransactions);

    //now adds the new transactions we're about to explore to the
        explored set
    exploredTransactions.addAll(thisAddressesTransactions);

    //all addresses linked directly (1 transaction hop away) from this
        address
    Stream<Address> linkedAddressesStream =
        getAddressesLinkedByTransactions(thisAddressesTransactions.
        parallelStream(), start, end);
    Set<Address> directlyLinkedAddresses = linkedAddressesStream.collect(
        Collectors.toSet());

    //all addresses linked transitively (2 transaction hops away) from
        this address
    Stream<Set<Address>> transitiveAddressStream =
        directlyLinkedAddresses
            .stream()
```

```
            .map(linkedAddress -> transitiveInputClustering(linkedAddress
                , exploredTransactions, start, end));
    directlyLinkedAddresses.addAll(transitiveAddressStream.flatMap(Set::
        stream).collect(Collectors.toSet()));

    return directlyLinkedAddresses;
}

private Stream<Transaction> getTransactionsForAddress(Address address,
    Date start, Date end) {
    return address.getOutputs()
            .parallelStream()
            .map(outputShell -> findOutputNode(outputShell.getOutputId(),
                 start, end))
            .filter(outputNode -> outputNode.getInputsTransaction() !=
                null)
            .map(outputNode -> outputNode.getInputsTransaction().
                getTransaction())
            .map(transactionShell -> findTransaction(transactionShell.
                getTransactionId()))
            .filter(transactionNode -> transactionNode.getInputs() !=
                null && transactionNode.getInputs().size() > 1);
}

private Stream<Address> getAddressesLinkedByTransactions(Stream<
    Transaction> transactionStream, Date start, Date end) {
    return transactionStream.flatMap(tx ->
        getAddressesLinkedByTransaction(tx, start, end));
}
private Stream<Address> getAddressesLinkedByTransaction(Transaction
    transaction, Date start, Date end) {
    return transaction.getInputs()
            .parallelStream()
            .map(InputRelation::getInput)
            .map(inputShell -> findOutputNode(inputShell.getOutputId(),
                start, end))
            .map(Output::getLockedToAddress);
}
```

8.4.1 Challenges

On-demand clustering can become unsuitable for extremely large clusters; for example, an address belonging to the wallet of SatoshiDice.com `1LaM2aDLEP49kLbE6y2hvnbrP3agbMwEHb` would belong to an extremely large cluster of many thousands of addresses, and clustering on-demand would take far too long to provide an acceptable user experience (we experimented with the above address and killed the request after 30 minutes had elapsed). For addresses belonging to large wallets like the one above, we attempt to cluster using the entity tagging information obtained in section 5 rather than using the *multi-input* heuristic. Since this data is stored in the database, it is quick to traverse the `HAS_ENTITY` relationships from a known address to find all of the addresses in the cluster. Therefore, we prioritise providing entity clustering information over *multi-input* clustering in order to provide as much valuable information over a shorter time; improving user experience.

For those addresses that do not have a link with a known entity, but exist as part of large address clusters, we truncate the search to a limit N that is defined by the user. The user can increase and decrease this limit, including disabling it completely, when initiating the search [see more on this in section 9]. Therefore, the algorithm will halt once N addresses belonging to the cluster have been found. This is a trade-off between usability and utility; the truncation makes the feature more usable since it provides more reasonable response times, however it does not provide a complete result. The user can still obtain a complete result by disabling the limit, albeit at the risk of poor user experience.

8.5 Clustering using raw CSV data

The challenges experienced in previous implementations were largely due to the necessity to write many new `HAS_ENTITY` relationships between address nodes to the already populated Neo4J database. If we were to circumvent this by calculating these relationships upfront before the bulk import into Neo4J, we would benefit from the powerful Neo4J Bulk Import tool making lighter work of writing these many relationships.

Once the relationships exist between clustered addresses in the graph, it will be efficient to query the database for addresses in the same cluster of a particular address; Neo4J is designed to traverse a large number of relationships extremely quickly.

Each entry in the CSV file will have the format below where `ADDRESS_IN_CLUSTER` and `ANOTHER_ADDRESS_IN_CLUSTER` are two Bitcoin addresses and `INPUTS_SAME_TX` is the name of the relationship.

```
ADDRESS_IN_CLUSTER, ANOTHER_ADDRESS_IN_CLUSTER, INPUTS_SAME_TX
```

The outline of the algorithm is as follows:

- for each Bitcoin transaction tx:
  - Fetches inputs ins of tx
  - Maps ins to the addresses the input is locked to and generates a set of addresses as which input the transaction
  - Writes a new CSV relationship entry for every pair $a1, a2$ in as where $a1! = a2$
- The new relationship CSV files are then used in the Neo4J import tool to re-import the entire database, now with clustering information represented in the form of relationships between address nodes.

We leveraged concurrency in the implementation of the above algorithm. We implemented the algorithm in Java, parallelising the work by splitting the transactions to be iterated through into batches, so the processing of each batch can be delegated to a separate thread. We addressed possible race conditions and file lock contentions by having each thread writing to its own output CSV file.

Chapter 9

Investigation Tool: *Radar*

Our investigation tool *Radar* is a single-page web application developed using Angular & Typescript.

9.1 Technology Choices

9.1.1 Angular 6 & TypeScript

Angular 6 provides excellent tooling for scaffolding out a wire-frame application and scaffolding out individual components. This is achievable using the `@angular-cli` tool where a command as simple as `ng new my-app; cd my-app; ng serve` will have a wire-frame web application up and running. This was a lucrative feature of Angular for this project since it would help expedite the web-application setup process.

Angular 6 applications are written in TypeScript; a statically typed superset of JavaScript. Using a statically-typed language is a personal preference, where possible, in order to allow for some compile time type-checking and easier refactoring.

Angular supports *RxJS*, an asynchronous programming library. *RxJS* provides an alternative approach to asynchronous requests; *Observables* are used rather than using Promises in JavaScript, which can only be subscribed to once. *Observables* can be subscribed and listened to indefinitely and can provide an unlimited number of data updates/messages. This is particularly useful for this application since every interaction the user makes with the graph will initiate an asynchronous request to fetch the data to render further nodes/information. These requests may be made concurrently too, so having a single *Observable* to listen for all new 'block data' or 'transaction data', for instance, greatly simplifies the implementation.

Alternatives to Angular

React is another solution to building single-page web-applications that is popular in the web development community. However, React is a library whereas Angular is a fully-fledged MVC framework providing far more 'out-the-box' functionality. React only provides the 'V' (the view) and will require several other libraries to fill in the model ('M') and controller ('C') components of MVC. Due to our familiarity with Angular, and relative unfamiliarity with React, there would potentially be a steep learning curve required to adopt React for this project. Angular would allow us to build and iterate fast since much of the basic functionality is provided by Angular 'out-the-box'.

9.1.2 D3

D3.js is a JavaScript library that can be used for generating SVG visualisations in the browser. D3 has no direct connection to Neo4J, unlike some other tools. The data will be retrieved by making HTTP calls to *Loran* and will involve further steps to fetch the data to render; however, this is advantageous as it will provide the flexibility to customise *Loran's* implementation to support complex features or to potentially use several data sources to orchestrate a response.

Alternatives to D3

There exist some visualisation tools that have embedded connections to Neo4J. We initially considered these tools as a possible simplification of the visualisation element of the web application; however, these tools such as Neovis.js and Popoto.js seem too restrictive in that their data format must align with the data format in Neo4J. They appear to be very suitable for visually mirroring the data in Neo4J, but we believe for this project, with the scope for many visual features with data persisted in locations other than Neo4J, this will prove to be too constraining. As for other JavaScript libraries similar to D3 with high customisability, we chose D3 due to the abundance of documentation, community support and example projects that D3 has.

9.2 Implementation

The most significant features provided by Angular that we used throughout the project were:

- **Components & Templates** - specifically, two-way data binding. The template will render the data defined in the component; changes to this data will lead to the template dynamically re-rendering to display the updated value. The component can listen to event hooks on the template, such as a click of a button or a value inserted, to retrieve user input/interactions from the template.
- **Services** - a way of passing data between components internally in the app and also used to retrieve data from external sources. The service 'InvestigationService' is largely used to communicate data across the components. For instance, the search component supplies the investigation service with search result data. This is published to an observable, which is being listened to by the investigation component which renders the new data as it arrives. Services help with separation of dependencies between components that should not need to know about each other.
- **Dependency Injection** - Injectable components (such as services) are requested by components that require them, rather than creating the dependencies them-self. This improves efficiency and modularity.

9.2.1 Routes

There existed two main routes within the application:

- `/search` : Loads the search component
- `/investigation` : Loads the investigation component (only if navigated there from the search form, otherwise it re-directs back to `/search`).

9.2.2 Architecture

The overall architecture of the web application can be seen in figure 9.1. The components shown in the figure have the following roles:

- **Search Component**: Renders the search forms. Validates users input, performs requests to fulfil searches using a Bitcoin-specific service component. Provides search results to Investigation specific service component.
- **Investigation Component**: Subscribes to the many data streams of the Investigation Service and coordinates adding new nodes and links to the graph.
- **Graph Component**: Listens to updates of the node/link data and updates the graph when data changes and initialises/re-starts the simulation when required.
- **Link Component**: Exists simply as a template for the Link SVG on the graph. Some logic around showing Output values in various currencies and timestamps.
- **Node Component**: Handles user interactions with the nodes on the graph. Identifies hover actions and double clicks and communicates accordingly with the services to inform other components of the actions. E.g., On hover, passes the node information to the App Service for displaying in the node data component. Additionally handles the pulsing animation while a node is requesting data to render its neighbours.
- **Add Node Component**: Provides a form for adding custom nodes to the graph. Validates form data and publishes new data to the Investigation service.
- **Add Link Component**: Provides a form for creating a link between custom nodes in the graph and any other node. Validates the form and publishes new data to the Investigation Service.
- **Node Data Component**: Responsible for rendering an overlay component displaying information about the node currently being hovered over. This data is published to an Observable in the App service and is subscribed to in order to receive updates.
- **Draggable Directive**: Applies D3's draggable behaviour to a node.
- **Zoomable Directive**: Applies D3's zoomable behaviour to the graph.
- **Investigation Service**: Mainly supplies data to the investigation component using Observables. For example, an observable to publish address data looks like this:

```
private addressData = new BehaviourSubject(null);
currentAddressData = this.addressData.asObservable();

supplyNewAddressData(newAddressData: Address) {
    this.addressData.next(newAddressData);
}
```

Then in the investigation component, a subscription to the observable will look like this:

```
this.investigationService.currentAddressData.subscribe((
    newAddressData: Address) => {
     //handle data update
});
```

- **Bitcoin Service**: Uses Angular's HTTP library to make HTTP requests to *Loran* to fetch new data. The calls return an observable with generic type corresponding to the data model it will populate. For example, a call to fetch an address node is:

```
return this.http.get<Address>(this.serviceDomain + "/bitcoin
    /getAddress/" + address + this.buildQueryParams());
```

- **App Service**: Uses observables to facilitate the transfer of data between node components and node data components.
- **D3 Service**: Handles calls to D3 to implement Draggable and Zoomable behaviours.

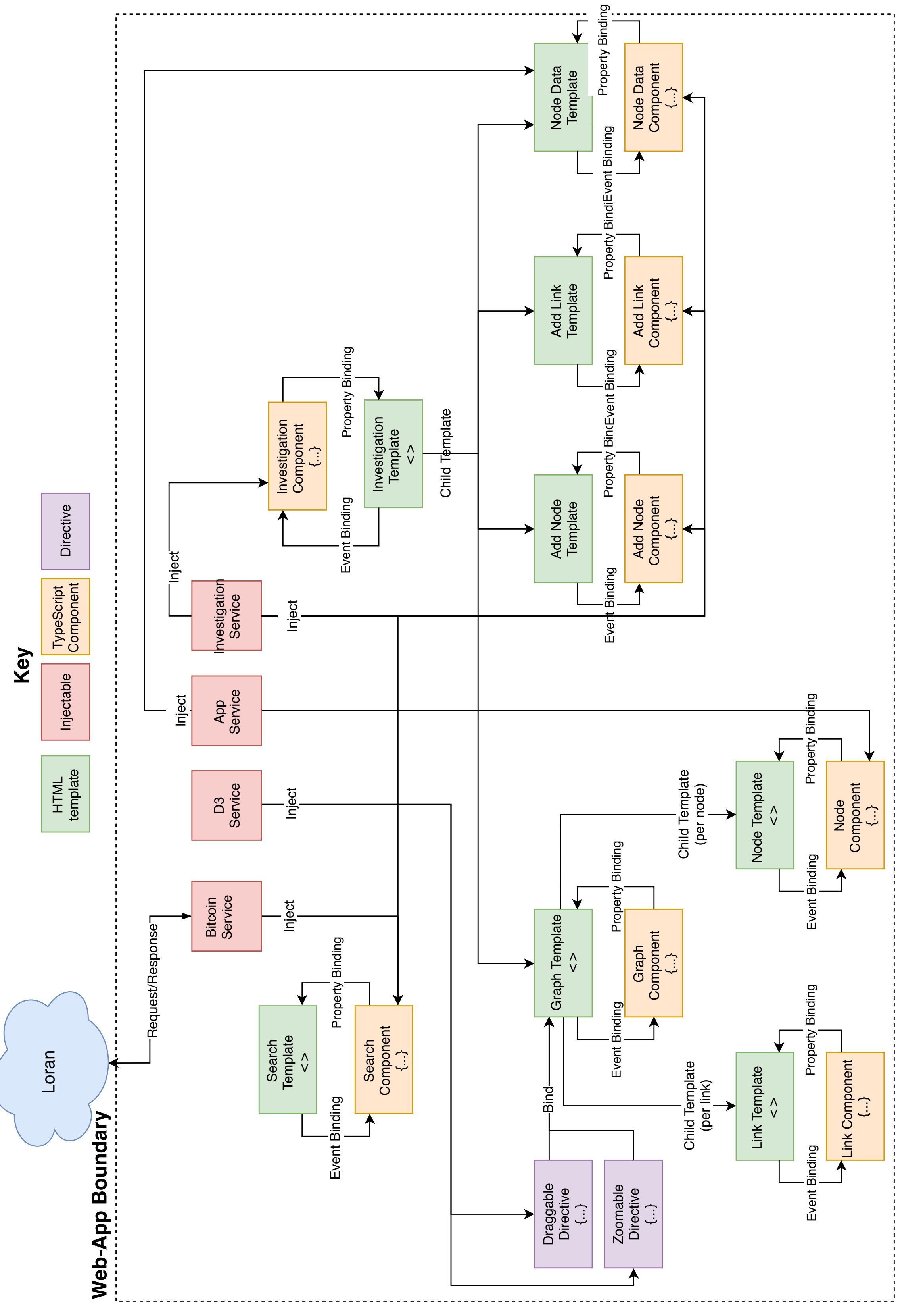

Figure 9.1: Architecture of the Web Application

9.3 Features

9.3.1 Search by Address

An investigation can be initiated by searching for a particular Bitcoin address. Submitting the search form as shown in figure 9.2, loads the investigation view displaying the address and its immediate neighbours as nodes in the graph, also detailing the relationship types each link represents, as shown in figure 9.3.

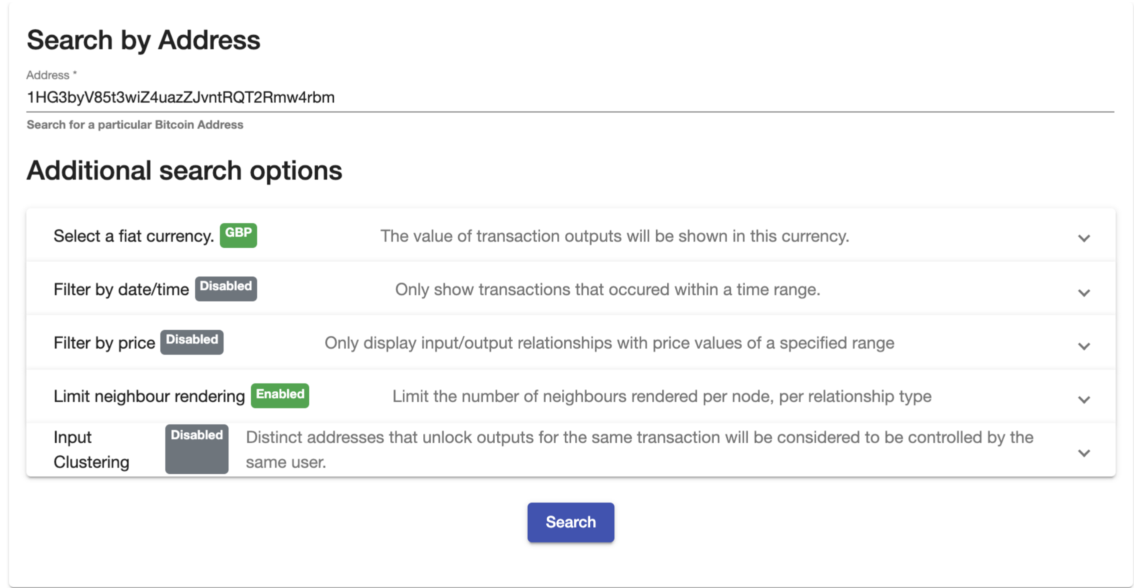

Figure 9.2: A screenshot of the address search form feature.

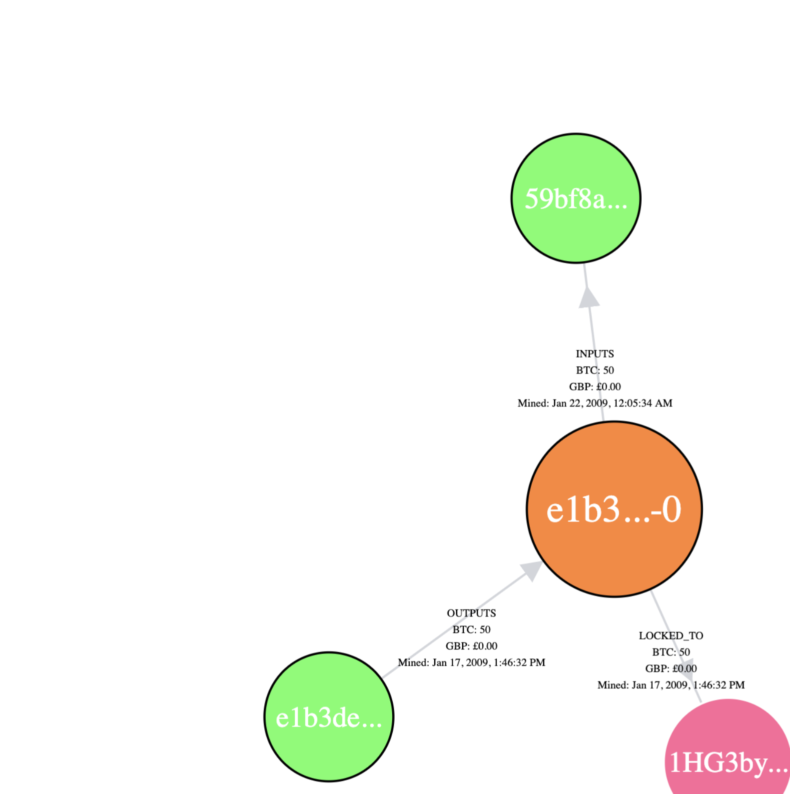

Figure 9.3: A screenshot of the search result produced by the search shown in figure 9.2

9.3.2 Search by Entity Name

In order to facilitate investigations that relate to thefts of popular exchanges, we provide the functionality to initiate an investigation using an entity name. If the entity name exists in the database (i.e. if it was collected in section 5 from walletexplorer.com), then *Radar* will be able to show a bigger picture of all the outputs/transactions associated with the entity, rather than just a single address it uses [see fig 9.5]. This will provide a better picture of the true transactions associated with an entity at a given date/time.

Begin investigation with an address

Search Type: Entity Name

Address or Entity *: Bitcoinica.com-old

Search by a entity name (e.g wallet/service)

Additional search options

Select a fiat currency. GBP	The value of transaction outputs will be shown in this currency.
Filter by date/time Enabled	Only show transactions that occured within a time range.
Filter by price Disabled	Only display input/output relationships with price values of a specified range
Limit neighbour rendering Enabled	Limit the number of neighbours rendered per node, per relationship type
Input Clustering Enabled	Distinct addresses that unlock outputs for the same transaction will be considered to be controlled by the same user.

Search

Figure 9.4: A screenshot of the search form when searching for the entity 'Bitcoinica-old.com'

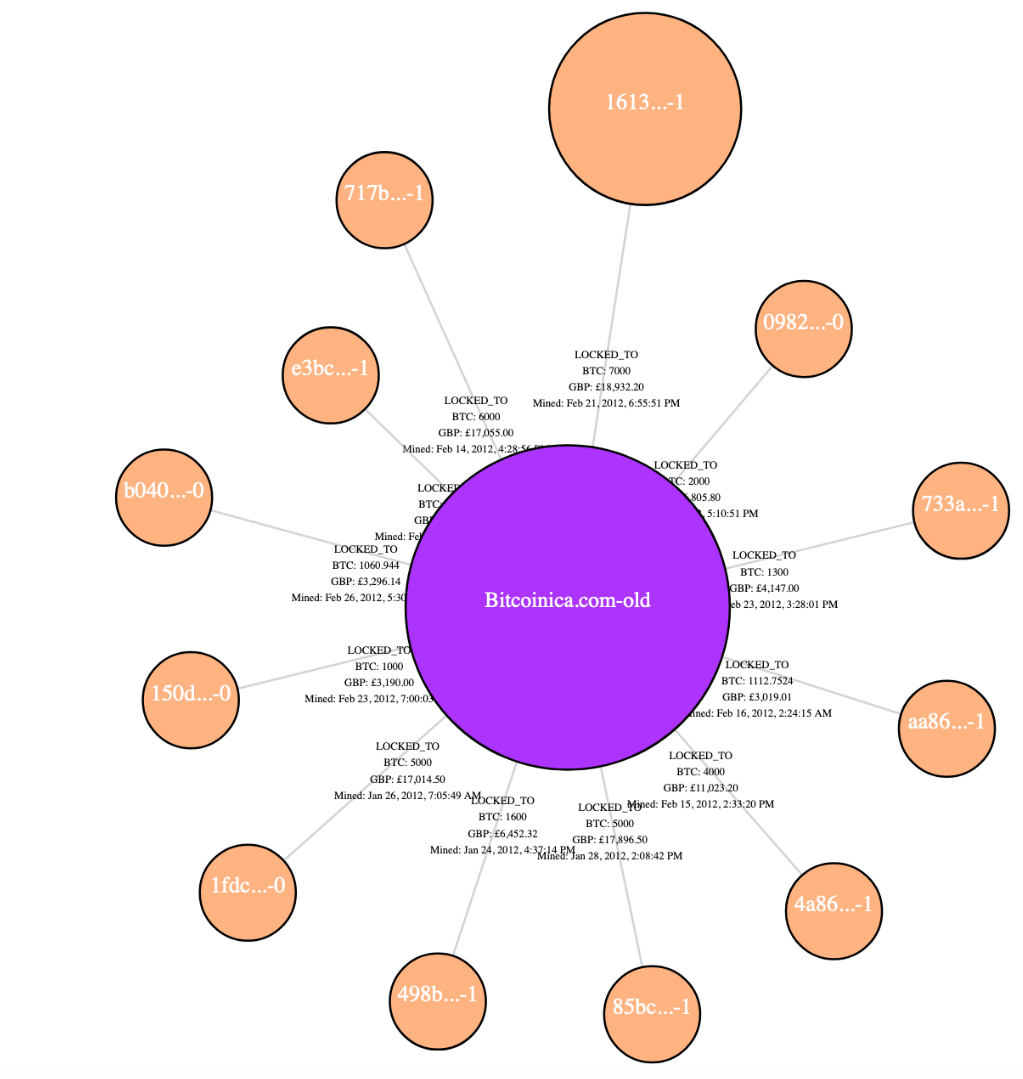

Figure 9.5: A screenshot of *Radar's* result when searching for the 'bitcoinica-old.com' entity

9.3.3 Node Information on Hover

Information associated with each type of node is displayed when a node is hovered over. For example, hovering over a block will display the time it was mined, its hash and the historical exchange rates at the time of mining (see fig 9.6). The node being hovered over will expand in size to visually indicate which node the information is being displayed for. There is also an option to dismiss the information box using the cross symbol in the top right corner.

Additionally, clicking on each field in the node information box automatically copies that data to the users' clipboard. This further improves the user experience as there will exist fewer steps to copying data (such as long ID's or hashes), which may be needed for transfer to another tool for further investigation. The data copied will also be copied without the extraneous surrounding data such as labels, currency symbols or data formatting; specifically, selecting dates will copy the epoch time in milliseconds to the clipboard.

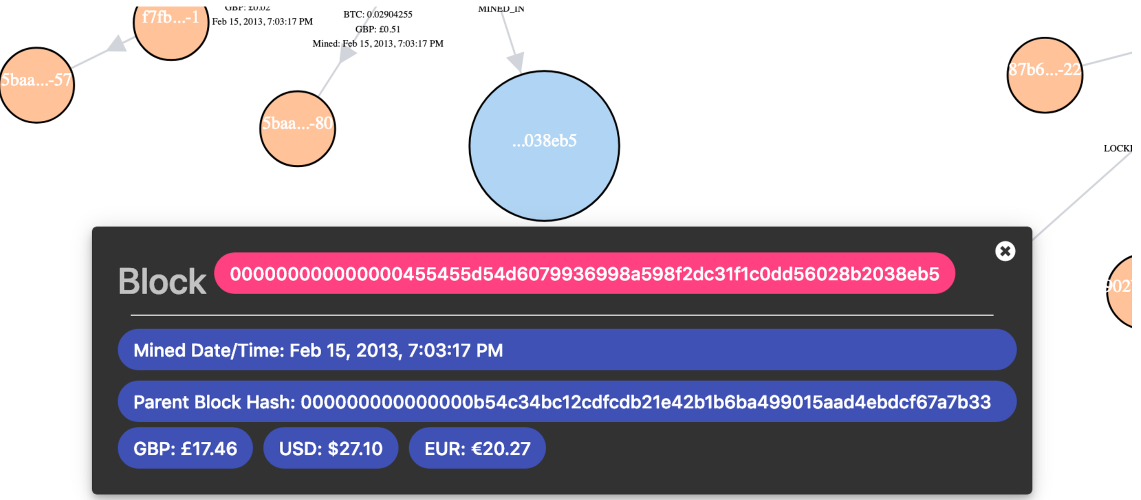

Figure 9.6: A screenshot of block information being displayed when hovering over a block node

9.3.4 Link Data

A primary use of *Radar* will be to investigate the flow of funds; it is important to make this information visible and digestible. Therefore, each link representing a flow of funds (between `OUTPUT` and `TRANSACTION` nodes) displays the value of the funds in BTC and whichever fiat currency the user selects as their preference when searching (GBP, EUR, USD). The link data also provides a timestamp representing the time the transaction was included in the blockchain; this provides context to the value of the output with respect to their fiat currencies.

The screenshot in figure 9.7 exemplifies the importance of a fiat currency conversion: the same unit of 0.5 BTC is output by transaction `432f18...`[1] as is input into transaction `96455e...`[2], however their equivalent fiat value in USD is very different at $1,688 when the output is produced and $2,102 when the output is spent. The timestamp will also provide context

[1] 432f18aa46626934c45805046f9c9791fb60bd41da1eb951b47f73bb3b8c7484
[2] 96455e8b6a877f273d14aa13ecb1971577c0d17716b7028c9809c7ca3e1f6e3c

around the timeline of this transfer of funds, in addition to providing an explanation for vast differences in fiat currency value of a particular output.

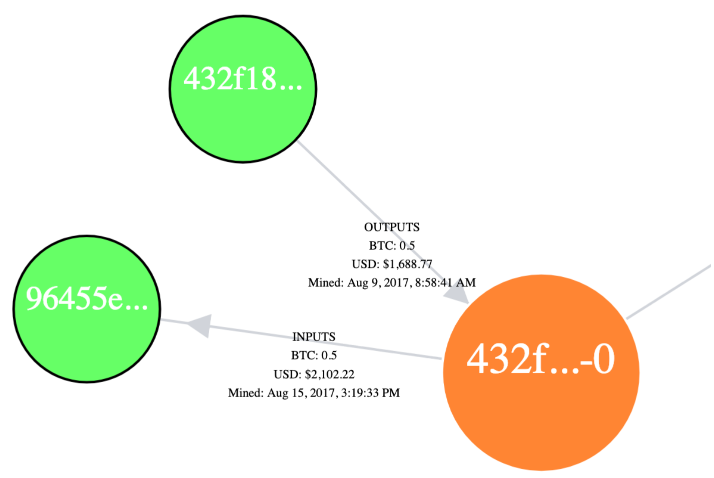

Figure 9.7: A screenshot link data between two transaction nodes (green) and an output node. Transactions IDs are: `432f18aa46626934c45805046f9c9791fb60bd41da1eb951b47f73bb3b8c7484`, `96455e8b6a877f273d14aa13ecb1971577c0d17716b7028c9809c7ca3e1f6e3c`

9.3.5 Traverse the Graph

Double-clicking on a node initiates a request to add that node's immediate neighbours to the graph. Once the request receives a response, the node's neighbours are added to the existing graph as new nodes and the relationships between the nodes will be represented with new links. Traversal of the graph is possible through the repetition of this process by double-clicking any of the new nodes that were added to the graph.

Sometimes the time between initiating a request and receiving a response isn't near instantaneous, perhaps due to demand on the database or a greater response size. Therefore, to feedback to the user that a request has been initiated, preventing them from issuing multiple requests or incorrectly concluding that a node has no neighbours, a node will pulse in size while a request is pending.

Additionally, when interacting with the graph, it may not be immediately clear which nodes have already been double clicked and had their neighbours loaded; this could be the case for nodes with their neighbours hidden due to filtering options. Therefore, nodes will have a thick black border until their nodes have been expanded (see both green transaction nodes in fig 9.7) and will have their border removed once expanded (see the orange output node in fig 9.7).

9.3.6 Link Dependant Colour and Size of Nodes

To indicate the differences in a node's connectivity among the many other nodes in the graph, a node's colour and size adjusts based on the number of links it has outgoing and incident on it. The rate at which a node's colour and size changes is normalised by the total number of links in the graph; a node with 30% of the links in a graph of 10 links will have the same size/colour as a node with 30% of the links in a graph with 100 links. This feature helps with identifying potentially more important nodes in a graph, and also assists with visual arrangement; nodes with more links will have a greater circumference and therefore easier to arrange many neighbouring nodes around it.

An example of this can be seen in figure 9.8 where an address `1HJtS7...`[3] has more links than another address `1HeYaB...`[4] and therefore has a higher colour density and larger radius. The figure also exemplifies this for output node `b777...0` which has more links than the other output nodes and is therefore slightly larger and of higher colour density.

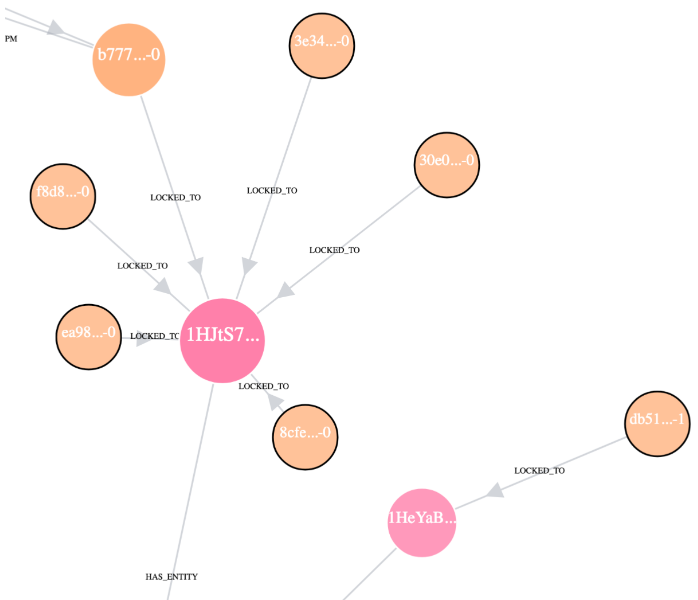

Figure 9.8: A screenshot of two address nodes, with a varying number of links each. Address IDs are: `1HJtS7wLaqdZRf1Cd4FfJDEWL17VJVDcm2`, `1HeYaB7gntUXQBtLaeJmqWzAdFf2PWMEfZ`

9.3.7 Selecting Fiat Currencies

The Link Data feature, as shown in figure 9.7, displays the fiat currency conversion of output values. The fiat currency displayed is configurable from the address search form, as shown in figure 9.9. See figure 9.10 and 9.11 for the graph differences when selecting USD or GBP.

[3] 1HJtS7wLaqdZRf1Cd4FfJDEWL17VJVDcm2
[4] 1HeYaB7gntUXQBtLaeJmqWzAdFf2PWMEfZ

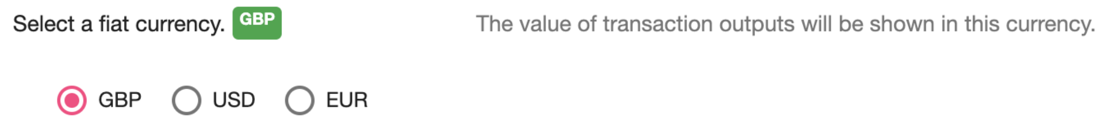

Figure 9.9: A screenshot of a radio button input field of the address search form, allowing one of three currencies to be selected

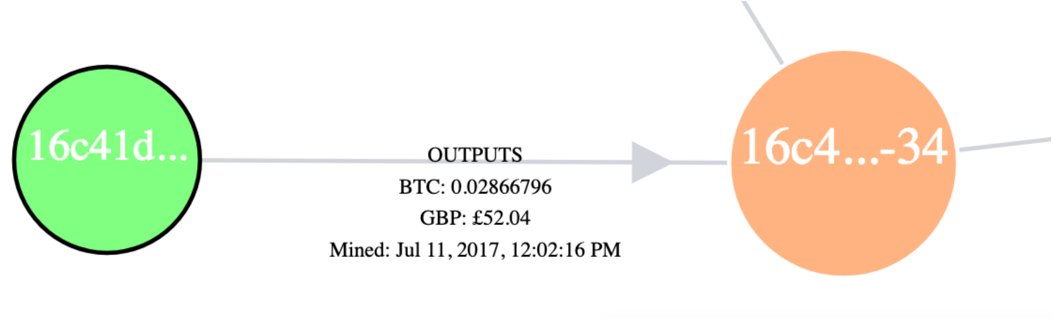

Figure 9.10: A screenshot of link data information displayed in GBP

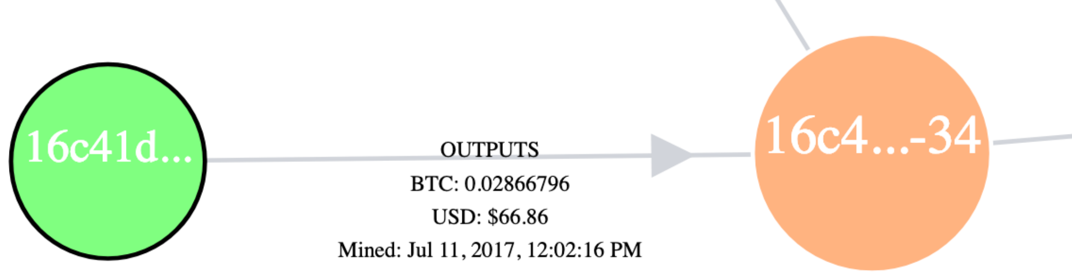

Figure 9.11: A screenshot of link data information displayed in USD

9.3.8 Filter by Date and Time

The date/time filter allows the user to specify that they only wish to see transactions/outputs that were mined in the time range specified. The option is disabled by default, but by selecting the toggle button to enable the feature, shown in figure 9.12, the user can adjust the start and end date/time for the filter using the calendar component for the date and slider for the time. The filtering will be further applied for any additional interactions (i.e. loading neighbouring nodes) the user makes when navigating the graph.

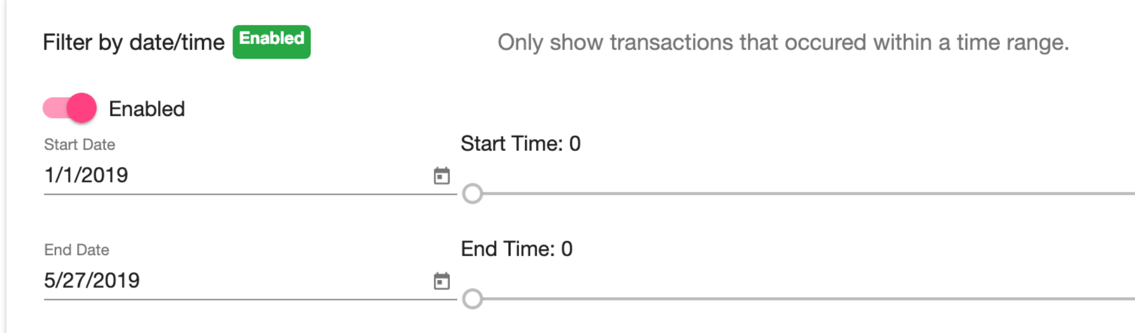

Figure 9.12: A screenshot of the date/time filter input enabled on the search form.

9.3.9 Filter by Value in Several Currencies

Similar to date filtering, transactions and outputs in the search results can also be filtered by their value. Their value can be filtered either in bitcoin (BTC) or any of the supported fiat currencies (GBP, USD, EUR). Once enabled, only transactions/outputs that have a value that lies in the filter range will be displayed.

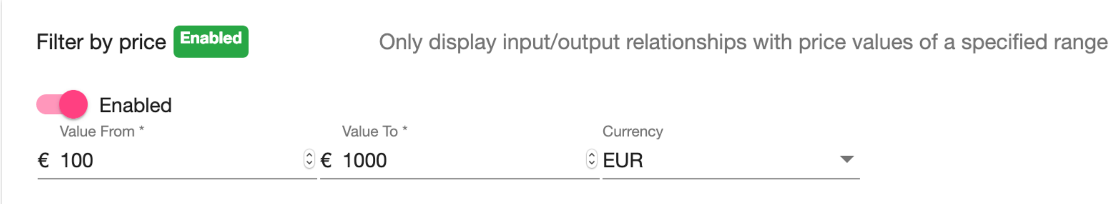

Figure 9.13: A screenshot of the price filter input field.

9.3.10 Limiting Nodes

Some nodes that are rendered on the graph may have a huge number of neighbours; such as a transaction with many inputs or an entity that has many addresses. If the UI were to attempt to render thousands of nodes using D3, the performance would seriously deteriorate, and the web application may become unresponsive. Therefore, in order to prevent this from happening as the default behaviour, the number of nodes that will be rendered will be limited, configurable using a field on the search form as shown in figure 9.14. The user can disable node limiting completely and is shown a warning when they do so (see figure 9.15). If they wish to adjust the number of nodes to render without completely disabling node limiting, they can easily do so using the slider on the search form.

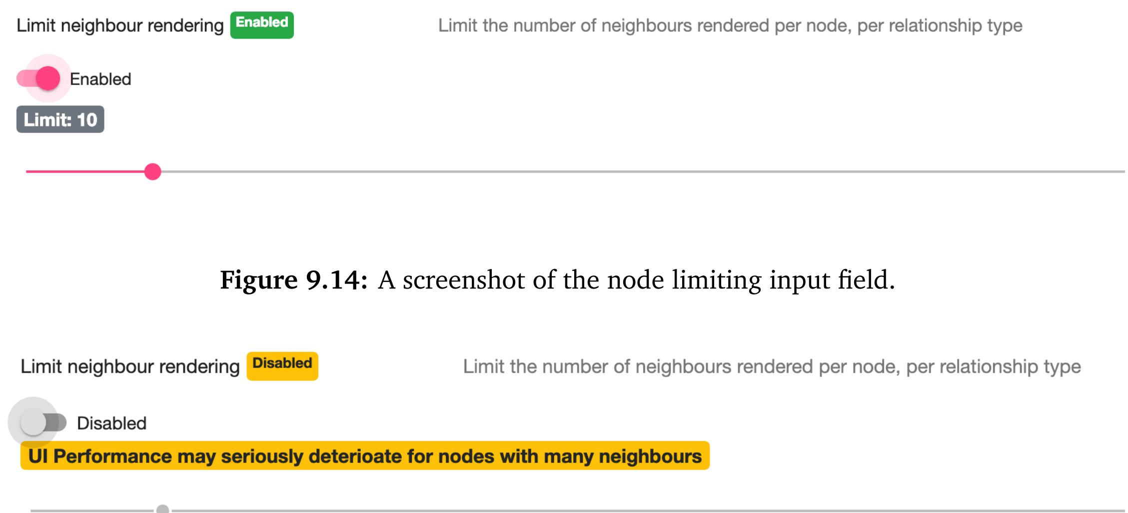

Figure 9.14: A screenshot of the node limiting input field.

Figure 9.15: A screenshot of the node limiting input field when disabled and showing a warning.

9.3.11 Enable *Multi-Input* Clustering View

The final option available on the search form is one labelled 'Input Clustering' which is a simple toggle, disabled by default, that the user can turn on when performing a search. Enabling this option will now display 'supernodes' rather than address nodes whenever addresses can be clustered; either through using the same-input heuristic as described in section 8 or using entity tagging data as described in section 5.

For example, an address that belongs to the wallet 'BitKonan.com' is `1GEp9Fui9XmyVhszPzFazdW3WHyt1adLR6`. When searching for this address, with the clustering option enabled, the search result will be shown with a large supernode for the wallet, rather than the individual address, as shown in figure 9.16. The output nodes connected to the supernode will be all of the outputs for each of the addresses belonging to the supernode. The addresses belonging to the supernode can be viewed by hovering over the node, where the node data component will display them as a list, as shown in figure 9.17.

For an address that does not belong to a known entity, but can be linked with other addresses using the *multi-input* clustering heuristic from section 8, then a similar supernode will appear, with all of the outputs associated with each address in the cluster rendered as neighbours.

Similar to limiting the number of outputs and transactions to render, the number of clustered addresses and linked outputs displayed is also limited when rendering supernodes. This prevents clustering 'on-demand' as described in section 8 from taking an unreasonable amount of time, while also providing the option to the user to disable the limiting from the search form and obtaining a complete response.

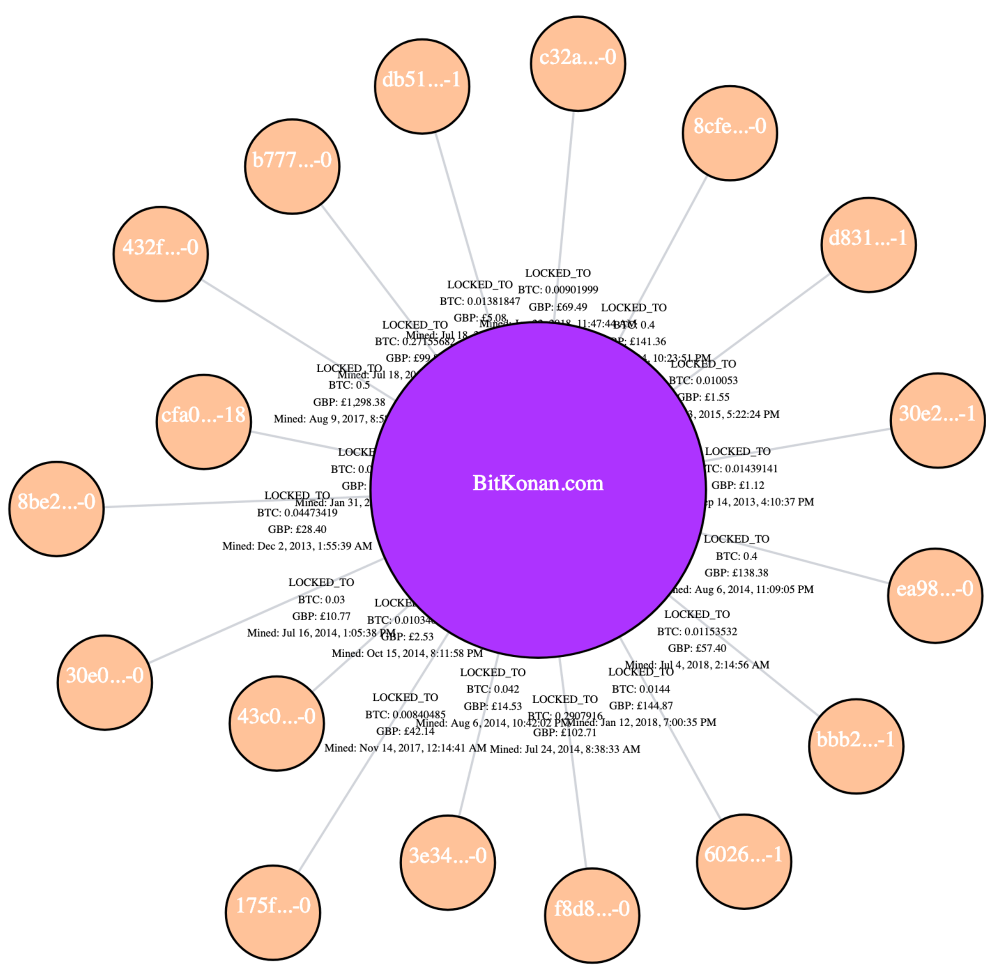

Figure 9.16: A screenshot of a supernode being displayed for address 1GEp9Fui9XmyVhszPzFazdW3WHyt1adLR6.

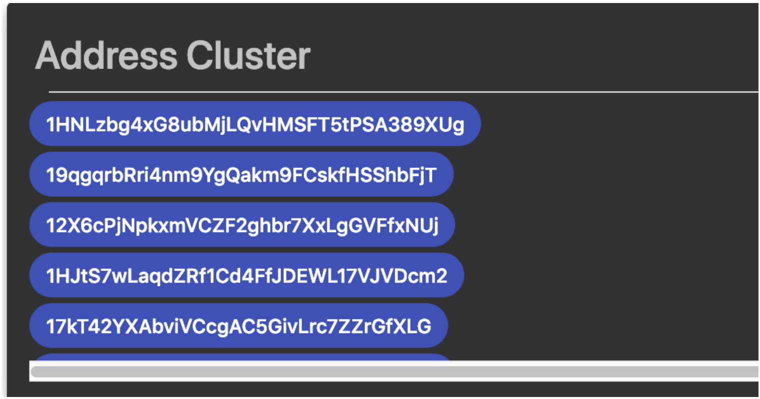

Figure 9.17: A screenshot of the node data component showing a list of addresses included in a supernode.

9.3.12 User Input Validation & Feedback

User input validation exists wherever the user manually inputs data across the web-app. Examples include user inputs required for searching by address, path finding, adding a custom node and creating a link. Examples of input validation in the application include highlighting fields in red and displaying a useful message, such as:

- On the Path Finder Form : No paths found between addresses *(if a path does not exist)*
- On the Address Search Form : 404, Entity not found *(if address does not exist)*

Additionally, to indicate loading states to the user, search forms display an animated loading bar while waiting for a request to return from *Loran*. This becomes particularly important for requests that have many linked addresses through clustering since some responses could take several seconds. To convey loading feedback to the user within the investigation view (the graph) when a user double-clicks a node and waits for its nodes to expand, we animate the node such that it pulsates until the request successfully completes and the new nodes are added to the graph.

9.3.13 Add Custom Nodes

It is possible to add custom nodes to the graph. The intention of this feature is the nodes will contain information to complement an active investigation; usually, this data will have an off-chain source. There exist various custom node types, based on the typical types of off-chain data that may exist in an investigation. One of these types is photographic ID, which provides the option to upload an image to be displayed in a custom node. The form for adding a new custom node of photographic id type can be seen in figure 9.18. The user can additionally choose to define an arbitrary number of custom fields in a key, value format.

Functionality also displayed in fig 9.18 is that the user has successfully uploaded a passport image. If a user uploads an image to a custom node, it will be displayed with its other information when a user hovers over the node. More on persisting custom data in section 9.3.16.

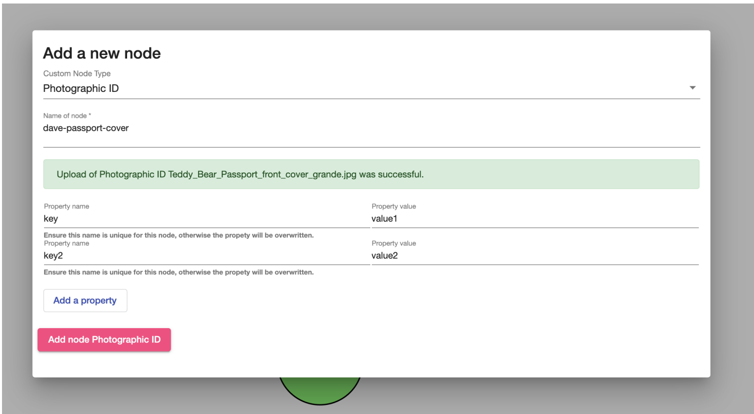

Figure 9.18: A screenshot of the form to add a custom node for passport data.

9.3.14 Link Custom Nodes to Other Nodes

Once a custom node exists, it can be associated with any other node shown on the graph by creating a link between them. The user can achieve this by hovering over the custom node, so that the node data view displays, as shown in figure 9.19, and clicking the 'Create a link' button. A modal form open which the user can submit to introduce the new link; the user must select the ID of the node they wish to create a link with. To help simplify this process, the target node form field will have a filtered autocomplete for all of the ID's that currently exist on the graph. The user can also change the direction of the link using the middle arrow button. The link name can then be any custom string defined by the user and will show on the link as shown in figure 9.21

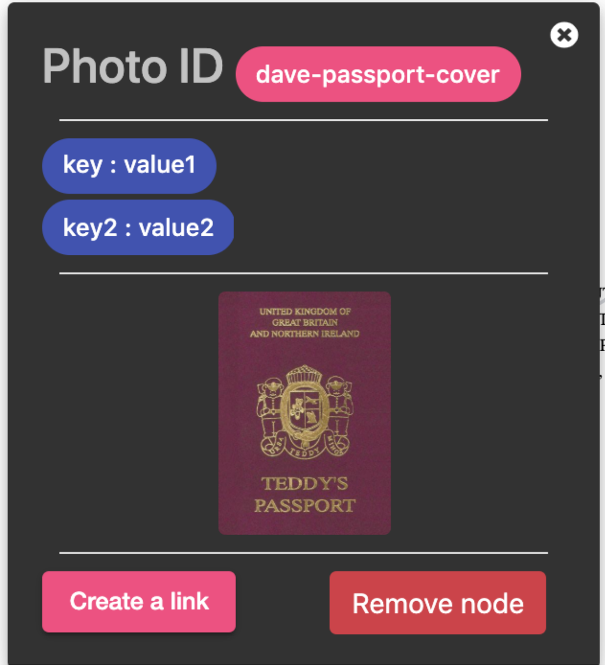

Figure 9.19: A screenshot of the custom node data component.

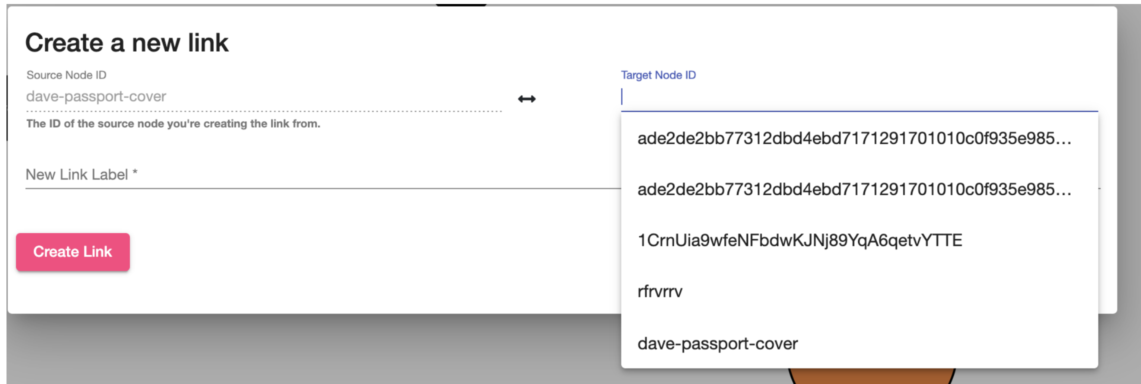

Figure 9.20: A screenshot of the form to add a new link from a custom node.

Figure 9.21: A screenshot of the graph after a new link `VERIFIED_IDENTITY` has been introduced between the custom node and an address node.

9.3.15 Path Finding

The path finder feature allows the user to search for the shortest path between any two valid addresses. The path can be of any length and include relationship types `INPUTS`, `OUTPUTS` or `LOCKED_TO`. This feature helps identify how two addresses may be linked through the transfer of funds. The result of the search is rendered in a graphical view, showing all nodes the path passes through and additionally highlighting the path using red links. Other neighbouring nodes extraneous to the shortest path are not shown by default but can be loaded using the double-click interaction. An example search result can be seen in figure 9.22. The two addresses shown are clearly connected by the funds they input into the same transaction.

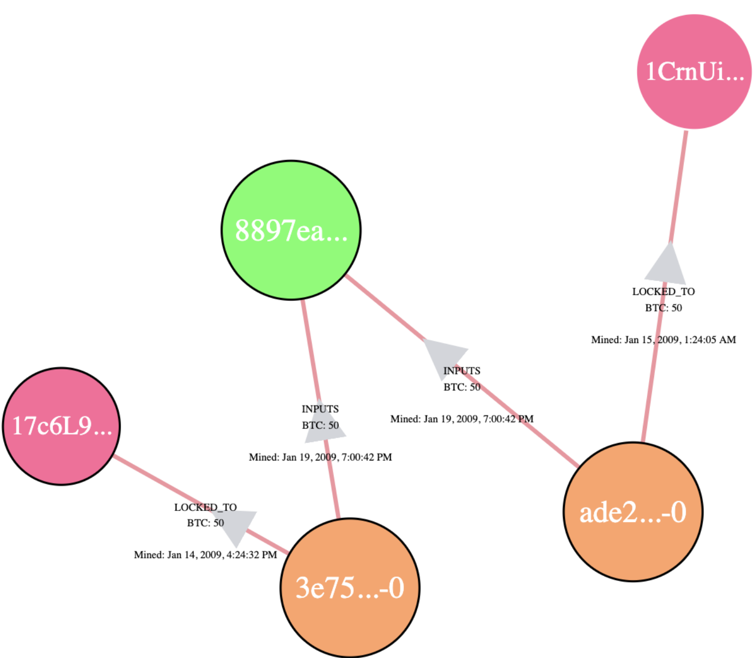

Figure 9.22: A screenshot of the result of a path finder search between addresses `17c6L9JUGVenn6CfqXuB93L3Tk8Tbzefui` and `1CrnUia9wfeNFbdwKJNj89YqA6qetvYTTE`.

9.3.16 Persisting Data

Instances where data needs to be persisted, such as when a user is uploading an image when creating a custom identification node, are handled by a simple web server implemented using expressjs.

We built a simple API to serve the purpose of uploading images.

```
POST /api/upload/{file}
```

This stores the file with its file name where the web server is located; allowing the files uploaded to be served as assets in the UI.

Chapter 10

Overall Deployment

This section describes how the previous sections, and the software components developed in each, come together to produce the final product.

Diagram in figure 10.1 shows the overall deployment of the software components built in this project. There are three main types of components in the diagram:

- **Active Components:** These software components are actively running as part of the final deployment of this project
- **One-Time Components:** These software components were manually run and used to build the foundations of the project, but do not need to be running for the deployed product. However, they are not completely redundant now. They will be required in the future, such as for adding new Bitcoin data, performing further clustering, fetching new price data etc.
- **External APIs:** These components were not developed as part of this project, but serve as a data source for the data used in this project.

All components are hosted on a VM on *Satoshi* [see 3.1].

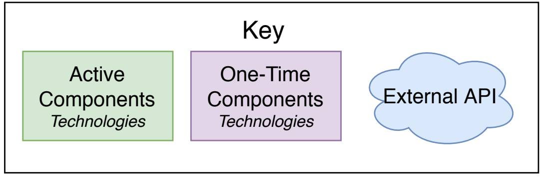

Figure 10.1: Overall Deployment Diagram.

10.1 Developing on Satoshi

The *Satoshi* machine (and VM) is only accessible from within the Imperial College network; yet for most of this project, we worked from outside of the network. This was facilitated using multi-hop SSH tunnel and a SOCKS proxy within the web browser; all browser requests on our local machine will then be forwarded using the SOCKS proxy through port 9999, then through an SSH tunnel to the Department of Computing's public facing shell server then through another tunnel (now inside the college network) to the VM on *Satoshi*. This setup can be seen in figure 10.2.

The commands used to set up this request routing were:

- On localhost: `ssh -l login -L 9999:localhost:9996 shell1.doc.ic.ac.uk`
- On DoC shell server: `ssh -l login -D 9996 satoshi.doc.ic.ac.uk -p 2222`

The end result was a development environment where we could navigate to localhost:8080 and localhost:7474 (for accessing *Radar* and the Neo4J UI respectively) in the browser on our local machine and for all requests to be routed to the VM on Satoshi.

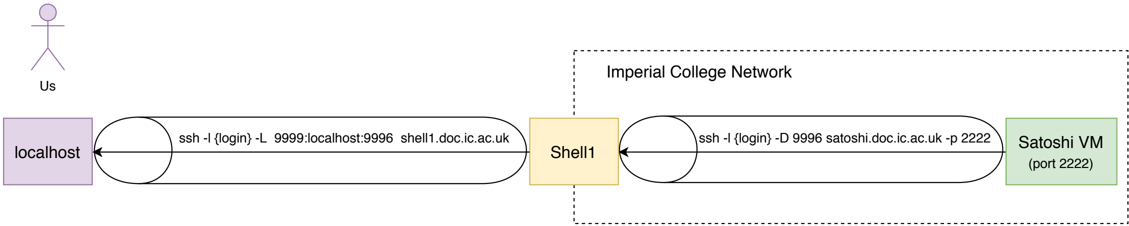

Figure 10.2: Proxy setup for development on Satoshi from outside the Imperial network

Chapter 11

Evaluation

11.1 Meeting Investigators from Industry

In order to gain feedback from the target end-users of this tool, we arranged a meeting with Mat Stanley, a Detective Sergeant of a London Metropolitan Police cryptocurrency investigation unit and, in order to also gain a private sector industry perspective, Iggy Azad who is a senior investigator at Coinbase.

We arranged the meeting with the goal of obtaining qualitative feedback on *Radar* based on their domain knowledge and experiences investigating cryptocurrency crime and learn how they currently achieve this using existing tools. We used this opportunity to find out more about the offerings of the proprietary software on the market that they have had experience with, such as ChainAnalysis, in order to compare the strengths and shortcomings of existing proprietary solutions.

11.1.1 Successes and Weaknesses

From the meeting with Mat and Iggy, where we demonstrated the current functionality of the tool, we obtained qualitative feedback such that we were able to identify areas of success and areas of weakness in the current implementation. These areas are summarised below:

- **Weakness**: The different types of nodes aren't very clear (*Implicit: We had to explain what the different types of nodes represent*).
- **Success**: The date and price filtering functionality is often very valuable to investigators. This is often particularly useful when tracing funds through a mixing service [see mixing services in background section 2.2.1].
- **Success**: Displaying the historical exchange rate is also very important to investigators; this enables them to understand better the true value of the flow of funds, which may be otherwise difficult without historical exchange rates due to the volatility in the value of Bitcoin.
- **Success & Weakness**: The Path Finding feature is very useful in investigations, yet not a feature provided by some of the software products they currently use. However, It would be useful to be able to type the name of a wallet rather than a specific address and the path finding functionality to be extended to support paths between any two wallets or any wallet and specific address.

- **Success**: Clustering using the wallet information obtained in section 5 will prove extremely useful; their ChainAnalysis software also provides this data.
- **Success**: The ability to toggle on/off clustering heuristics is very desirable and not provided by several other solutions.
- **Success & Weakness**: Very common usage is to add additional information to compliment an investigation, so being able to do so through the custom node feature is very useful; however, adding information such as Photographic id is not usually required at this stage of the investigation. The tool will be used in the run-up to issuing a warrant to obtain identification from exchanges (who should have it due to KYC [see Know Your Customer in background section 2.9). Additionally, it would be more intuitive to store this data with a particular node, rather than existing as a separate node itself.
- **Weakness**: Secure data storage not addressed by current deployment: If storing investigation specific data, a secure solution is needed such that only the user could access it.

11.1.2 Desirable Features

Further to our conversation with Mat and Iggy, we were able to curate a list of features that would be useful to someone wishing to investigate cryptocurrency using *Radar*. These features are:

- Support for several cryptocurrencies. Namely: Ethereum, Litecoin, Bitcoin, Bitcoin Cash, Dash
- Ability to save a graph state as an investigation and come back to it later
- Ability to export the data from an investigation, often in CSV format to provide evidence for cases - needs to be digestible by a jury.
- Ability to set up notifications/watches for state changes on entities - e.g. an address/wallet receives or spends funds
- Automatically detect and alert when an address re-surfaces in a new investigation that was seen in a previous investigation
- Ability to collapse nodes/remove nodes from the graph
- Offering a compliance/risk score with clustering information - is the data verified? Does the data come from multiple independent sources?
- Enhance clustering of addresses by considering different types of addresses [see address types in background section 2.1.2]
- Incorporate address tag information - they often have some significance, e.g. tags may relate to a gamer tag which can be searched for in other datasets. Also, vanity addresses often have significance, may include parts of their name which can be used to help link addresses [see vanity addresses in background section 2.1.3]

11.1.3 Additional Data Sources

In addition to the additional features, Mat and Iggy also helped us identify additional resources available that could potentially be used as data sources to feed functionality for the tool. These data sources are:

- oxt.me: Useful for looking at multi-signature transactions
- bitcoinswhoswho.com: Check addresses against reported scams or see if the address has associated tags
- Shapeshifter: Given a TXID, tells you where the coin went (which currency?). Also has a free API.

11.2 Performance

In this section, we gather quantitative measurements on metrics such as query response times in order to have measurements on the typical users' experience when making various requests.

11.2.1 Individual Cypher Query Profiling

In this section, we use the Cypher command `PROFILE` to profile queries in isolation for randomly selected data (in isolation meaning the database is not currently handling any other queries).

Through profiling, it became clear that Neo4J would cache recently executed query search results, and therefore repeating the same query would have a significantly reduced response time.

Selecting Random ids to Query

In order to prevent the Neo4J instance from caching request responses, potentially leading to unrealistic results in our experiments, the database must not have seen the address in a query before. Therefore, to retrieve the addresses to use in the profiled commands, we used another instance of Neo4J on a local machine to collect the ids to use. This will better reflect the performance of the queries that will be executed for the users' address searches when a cache hit will be highly unlikely. We executed the following query to retrieve the addresses to use: `MATCH(n: ADDRESS) RETURN n LIMIT 20` where 20 is how many ids we would like to use for profiling.

Profiling Node Retrievals

We performed profiling for 20 randomly selected ids for several types of nodes. For example, for profiling address nodes we used the Cypher query:

```
PROFILE MATCH(n: ADDRESS address:'address param') RETURN n
```

address	response time
14KTLEerB7uWPYZ5KcsxrJZwhFUDVwX9ZN	22ms
14NHfjeRs7vPFVKkAhQg8hWbFr1cemtP1X	17ms
14NNcfY2eFjG5YCC2DmgZfjj8kYmfAvWNs	151ms
14NvmcsMQsMmmWkjYQz4XWKFYFZVvjtwg1	16ms
14PWoVZf2zLP7Xdbpxa98c3VpuhqLgiDEh	17ms
14QFGVvDKJZEBMagGyDctfvboG2bgboBAU	13ms
14QyWYad5GsTmDvcJZD3d2Pjf5FUT1NyMn	19ms
14SJo9Ym8A4vEBBkozrAv8boQHxtDKozSd	34ms
14U5EYTN54agAngQu92D9gESvHYfKw8EqA	13ms
14UV4bnpdNH1AnhJYHzgGRRJbS7Z6t4Zh9	21ms
14UjUdVChndAh8r2yvNbXXhptb2qmrbmG2	16ms
14UyXKWBcVHYcqSpxYcmR9JLZu6pArwDrd	13ms
14VzDyJrL5FBM3vuWd2ngCmwkQAetN2zqy	22ms
14ZT7wtrwjAVGGAB6E8tdHg63q1GaTdCSY	19ms
14ZZFUVoNyXPJrssTgNfmB7LrrpNeSotQx	55ms
14bj9QJfyUWTcw5BubEccohmWz5XLYGG9x	15ms
14bn4VMSpRtb7EDUwpX2rnqC8WSHoHNyMn	16ms
14bz2YUvxEWAhYk7EkVznU3pJC36Lq9967	19ms
14c26aYYJ4K9Wfeh9Uo5KZbkSREzYcjx9i	14ms
14c5uQYWBFPfwwRpmMud8GGj7tFAfC7fE5	16ms
average	**26.4 ms**

We performed the same profiling for each other type of entity that is retrieved as part of the functionality for the site. These other entities are shown below along with their average response times for 20 random ids of each type. Each individual query response time is omitted for brevity.

Node Type	Average response time
`ADDRESS`	26.4ms
`TRANSACTION`	20.8ms
`OUTPUT`	23.55ms
`BLOCK`	10.90ms
`ENTITY`	9.45ms

The query profiling results show simple lookups all lay in the range of 9-30ms, which feels near instantaneous as a user waiting for a search result. There are slight differences notable between response times for address, transaction and output lookups compared to block and entity simply due to the differences in the size of the indexes that need to be searched; obviously, the output index will be substantially larger than the block index as there will be substantially more outputs than blocks.

Profiling Path Finding

We profile the underlying Cypher query used to implement the path finding feature. To do this, we select two random addresses $a1$, $a2$, similar to the previous section, and execute the query to find a path between $a1$ and $a2$. We select 10 pairs of random addresses and repeat the profiling 10 times. The results of this process are shown below.

start address	**end address**	**response time (ms)**
bc1qzzzc2q48adfzv8p-zd5nnjhuqxml8pjm46a8rsq	bc1qzzxn3pykntgxnne-8etmn4fuqt5hf3s2d5zwh0t	51737
bc1qzzzq52uf4uhyurd-wpqn96tethg8mmr295twne7	bc1qzzxw0ugl2whm7q0-2lnnnkvsjy0g78w8p4s7mse	470
bc1qzzyrfhgk9shmlrs-wwvg64x4awjl3jq3nql3aq5	bc1qzzz96yje7jxkwkk-4a8aed64m7dq7ek7wxmpsad	224649
bc1qzzyl9nfa20hwt26-mrvm346tyv2sk72j8qdjvdk	bc1qzzzupj2n80wk7tu-z662twmgdlppvlg25fqfpxy	100016
bc1qzzzr4xau2d88e35-952t7zvpujvsnlxh23qdga7	bc1qzzz26k5gkplu8la-jsg5kp0acyzpxkpp5ungcn0	44905
bc1qzzzlt8nd9f80a2u-c005kavr9dctmuwjhk3fgh7	bc1qzzz4zvemglwlhue-r56wq4nl2vrlmr7hslex23h	3045
bc1qzzyh94k2r5m4fdx-pa4yjnt8j8swjtkr49zt3gp	bc1qzzzelhfgx6tydat-00khrpmrr75d4ddf57zw0nk	1183
bc1qzzz72vr3lvz3967-fanykm4f6890u5gkqspw2pu	bc1qzzxvqvtga8trt0r-064sn5sam2p89uy2hmffzug	3291
bc1qzzxmgns300890vj-tvk6rty2weker8ejaythytc	bc1qzzzlnqktq3gtqfa-jet24828x839em2a6my0ttx	14924
bc1qzzygmd9klrqce6q-j0pfx2vzk763z8stq457zv5	bc1qzzzagxs2f5dtk2u-dx7q4zqc89n20hgzv5e0xw0	37134
average		**48135.4 ms**

As shown from the table above, the average response times reported when profiling path finding queries are magnitudes greater than for individual nodes.

11.2.2 Performance Under Load

Below we evaluate the performance of requests sent to *Loran* created in section 7. If we were to evaluate the performance of issuing requests individually and sequential, we would not obtain a realistic representation of the expected performance in a production environment. Therefore, we use Locust.io to simulate many users concurrently interacting with *Radar* and invoking requests to the API. We also use Locust to measure statistics regarding latency and counts of success/fail responses.

Using Locust, we define a set of tasks that a typical user may perform, with a time range in which a user may perform them in. We then run experiments which launch a 'swarm' of simulated users performing these requests concurrently in order to observe how the system performs under load; particularly inspecting failure rates and response times.

The tasks defined (which are translated to API calls) are:

- Search for a random address
- Search for a random output
- Search for a random transaction
- Search for a random block

- Search for a path between two random addresses

These tasks invoke a call to the relevant API endpoint to retrieve the data. In order to avoid the entities already being cached, the entities ids used to retrieve entities are randomly sampled from a list of 1000 possible ids that have not been used in previous experiments and are therefore highly unlikely to be cached.

Assumptions

When running the experiments, we make the assumption that each of the above tasks are carried out at random by a user with a delay of 5-9 seconds (range randomly sampled) before performing the next task.

10 Concurrent Users

Figure C.2 shows the requests per seconds (top graph) and the median (green) & 95th percentile (yellow) response times (bottom graph).

Observations:

- Failure rate of 0%
- Request rate of around 1 RPS
- As shown in figure C.2 the 95th percentile measurement will occasionally spike
- Spike can be directly attributed to individual requests to the shortest path endpoint [see fig C.1]

100 Concurrent Users

We attempted this experiment in order to observe how the system behaves under a significantly larger load. The statistics dashboard for this experiment can be seen in C.3.

Observations:

- At approximately one minute, there was a period where no responses were received (see the 'no data' label in figure C.3)
- At this time, requests began to return errors of 500 series
- Investigation led us to discover the cause was due to the connection pool for the database becoming full and the timeout for waiting for a connection to become free had elapsed
- The connection pool became saturated due to several connections being consumed by expensive path finding tasks
- The significant latencies of the path finding tasks (for those that did return) can be seen in figure C.4, and can be seen towards the end of the timeline in figure C.3 where the response time spikes to the 150,000 ms (150 s) range.

Clearly, invoking several concurrent path finding requests is problematic and will cause user requests (including non-path finding requests) to fail.

10 Concurrent Users (without Path Finding)

In order to observe how all other requests would behave without the path finding feature, we removed the path finding request from the users' behaviour.

Observations:

- A 0% error rate
- There are still some spikes, that can be attributed to a few requests for Output entities [see figure C.5 and C.6].

10 Concurrent Users (without Path Finding + with Node Limiting)

Clearly, fetching entities with an unbounded number of neighbours can cause issues, so we introduced the node limiting query parameter which is used to implement the 'node limiting' functionality as described in section 9.3.10.

Observations:

- 0% error rate as before
- Spikes now dampened due to node limiting
- Expected response rates for entire experiment [see fig C.7]

100 Concurrent Users (without Path Finding + with Node Limiting)

Observations:

- Now a 0% error rate due to the absence of path finding request
- Spikes dampened by node limiting [see fig C.8]
- Acceptable median response rates for the entire experiment
- Slightly higher 95th percentile response times compared with 10 users, due to greater load

Conclusion

From these experiments, we learnt that under significant load, the path finding queries are capable of causing a denial of service to other users of the system. Additionally, requests that retrieve entities without an upper-bound on the number of neighbours to fetch, request response times become less predictable. However, without path finding requests and with the default node limiting option enabled, the system will be able to handle considerable loads (up to 100 users) comfortably.

11.2.3 Blockchain Import

As described in section 6, running *Astrolabe* to import the Bitcoin Blockchain up to block 570,000 took **5 hours, 32 minutes and 54 seconds**, which was performed in a VM with the specifications described earlier in section 3.1.

The alternative approaches to performing this task are discussed in section 2.8. These tools have the following performance metrics:

- Bitcoin to Neo4J Tool: For block height 466,874, the GitHub page says it can take more than 60 days. Performed on Thinkpad X220 (8GB Ram, 4x2.60GHz CPU) [26].
- TokenAnalyst Approach: Achieve the download and database population in one working day (approx. 9 hours). They took 6 hours to fetch all bitcoin data using RPC, then convert it to CSV format. TokenAnalyst explain how they had 32 cores at their disposal and were able to perform the import in 1 hour 10 minutes [27].
- Blockchain2graph provides stress test information on their Github page, which achieves 48,000 blocks in 24 hours [29]. They do not describe the hardware for this benchmark data; however, as some indication of the lower bound of hardware used, their 'old' tests ran on an Intel Atom with 4GB RAM and 1 TB of disk space.

Existing Solution Comparison

With such different hardware and block heights used for each of the above benchmarks, it would not be useful to extrapolate the results above to the same block height in order to compare metrics. We would also need to make approximate estimates on expected performance improvements gained if equivalent hardware was used, adding further ambiguity to the meaningfulness of a quantitative comparison. Rather, such comparisons need to be made both qualitatively and pragmatically.

Some simpler comparisons can still be made. Clearly, the performance of our approach far exceeds the implementation of the Bitcoin to Neo4J tool. The approach uses transactional queries to add data to the database, which may explain the most significant performance issues with this approach. Neo4J's bulk import tool, used by our implementation, bypasses the transnational layer and is designed to heavily utilise parallelisation across multiple cores which vastly expediates this import process.

TokenAnalyst use the Neo4J import tool and can achieve a very impressive import time of **1 hour 10 minutes**; this is faster than our import time, but TokenAnalyst have the advantage of more powerful hardware with more cores, therefore allowing for greater parallelisation. This indicates that hardware is the bottleneck for our database population time.

Due to the relatively recent publication of TokenAnalyst's article describing their process, we can assume their import process will be to a block height that was recent and therefore similar to the block height of 570,000 we used for the bulk import process.

Since TokenAnalyst performed the job of populating a database with Bitcoin data in two core steps, first downloading then importing, as we did, we can also compare our respective times to download and write all Bitcoin transactions to CSV.

TokenAnalyst are able to achieve writing all Bitcoin data to CSV in 6 hours, compared *Astrolabe* taking 12 hours [see section 3]. Again, greater physical resources will have given them greater ability to parallelise the workload and achieve better performance. However, one weakness of *Astrolabe* is the necessity to manually intervene between sub-jobs (i.e. between the download of blocks 100k-200k and 201k-300k etc). This was done to mitigate the consequences of the overall job failing, however, this mitigation could be replicated in an automated fashion in order to reduce idle time across the download period.

11.3 Performing a Historical Investigation

As described earlier in section 2.6.4, there are several well-known thefts of Bitcoin exchanges, where vast sums of bitcoin were stolen. Using simple information available for these thefts, such as the wallet name and the date, we show how *Radar* can be used to provide a better understanding into the flow of bitcoin funds away from its rightful owner.

One example we use is the Linode Hacks which targeted *Bitcoinica.com* on the 1st March 2012. We used this wallet name (*bitcoinica.com-old*) as a starting point of the investigation, only looking at transactions for the date 1st March 2012.

Initially, we searched for the wallet name with a date filter only. This rendered the view shown in figure 11.1.
Obviously, the huge amount of information shown in figure 11.1 is not feasible to digest and understand; therefore we navigated back to the search form to start refining the filters to help better specify what we are looking for.

We first added price filters in order to show only the high-value transactions on 1st March; we filtered transactions to only show outputs that were spent of value more than $1000BTC$.

We then proceeded to inspect the outputs that were spent on 1st March that were of the greatest value [see fig 11.2]. By doing so, and expanding out several other high-value outputs spent on that day, we were able to identify transactions that spent a large amount of funds. We considered these transactions as suspicious.

The suspicious transactions we identified through this process due to the size of the funds they spent on this day were:

- `7b45c1742ca9f544cccd92d319ef8a5e19b7dcb8742990724c6a9c2f569ae732`: This transaction in total spent **20,555.1 BTC** - sending all **20,555 BTC** to one address and **0.01 BTC** to another. Using the neighbour expansion feature, we traced the bulk of the funds:
- `0268b7285b95444808753969099f7ae43fb4193d442e3e0deebb10e2bb1764d0`
- `a82ad85286c68f37a2feda1f5e8a4efa9db1e642b4ef53cb9fd86170169e5e68`
- `a57132e2cbc580ac262aa3f7bac1e441d6573f9633118bc48009618585a0967e`
- `901dbcef30a541b8b55fae8f7ad9917ef0754bda5b643705f3773e590785c4d3`

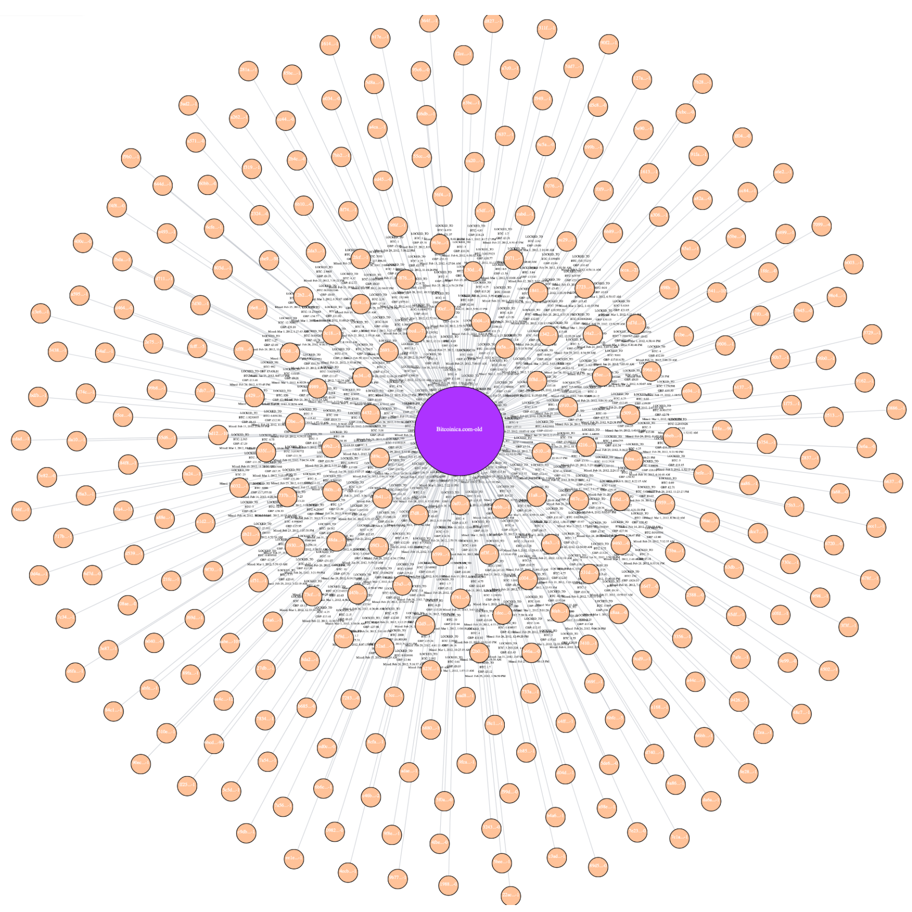

Figure 11.1: A screenshot of the view when searching for entity *Bitcoinica.com*, filtered by date on 1st March 2012 only

.

We were able to verify these transactions as suspicious by searching for information relating to these transactions and discovering the post [1] announcing this hack and the suspicious transactions identified at the time.

To attempt to simulate what an investigation may involve, we attempt to trace/understand the flow and destination of funds taken in transaction `7b45c1742ca9f544cccd92d319ef8a5e19b7dcb8742990724c6a9c2f569ae732`:

1. **20555BTC** to `1DMuVKe9PKpx3dbs2b2MnXuVmLfA4drHif`
2. **25000BTC** to `128u4nNS2DCbPk61aNAXLUTsHZt5FAEtit`
3. **24900BTC** to `18E4d3JtQNyUBdxQ4y8ck6RUKXEb7W4KA7`
4. **23900BTC** to `1D56cDkVmoNGw7YmewL5boFwHyzEgDpygE`
5. **23399BTC** to `1LJn39nzrwqM8MkevfSDCa1uVZCgJSTMtq`
6. **22399BTC** to `13Fd5f8yFZCAAbtKQiNcWUmdbUNpVpznA6`

[1] https://bitcointalk.to/index.php?topic=66979.0

7. **21899BTC** to 1HZHp77ftbAEy4PQc8wuAahvPnpZHKMfmc
8. **16899BTC** to 1Czhv7mDRNPMrzU9Ja7QkYSAvforGKQ2mv
   (a) **5000BTC** to 1Q3bsvTBcWF32Bt8FZgKAx7s43crvN9RVi
9. **16900BTC** to 1Q3bsvTBcWF32Bt8FZgKAx7s43crvN9RVi
10. **17000BTC** to 1BKF8kXHsNdQk6f5FognK4ZJCCBifE8ugm
11. **15476BTC** to 1MXe7zFRHx9KMwGKAGi2GvTbsHFDgVjcEw
12. **12544BTC** to 15GPUomoHVJbBsgY6eWdsKKaV3dywL5poj
13. **12044BTC** to 12F1GhUbVJvNW6ki1fRG1ZgeTkm2FixLLi

The main stream of funds make its way to 18ywk3t8XFtkEqMJt8hPG14D3gzTshB4YW where funds are now not spent until 4th August 2012. This seems to be the beginning of another chain, spending sums of 100-1000 BTC (usually repeated integer values such as 100, 200, 500 etc) in each transaction, each transaction having only two outputs until the main large funds reduce down to just over 1000 BTC with transaction fbab826df674b1e2ae3bedfc566f2692b8940ac1058605fe549753b34ebd8f0d. This seems to be the end of the peeling chain, as transactions now show characteristics more common to normal spending; outputs are spent gradually, rather than many in a very short period. Also, transactions have several inputs, rater than one or two and outputs are of non-integer, more random looking amouts.
The tool has clearly helped us identify the signature of a peeling chain; a visual representation of the peeling chain (showing the main addresses and flow of funds) can be seen in figure 11.3; the nodes of the same colour are addresses that were clustered into a supernode using the 'same-input' heuristic. This shows that many of the funds from the chain end up in a cluster controlled by the same user; therefore making the obscuring efforts of the peeling chain more transparent.

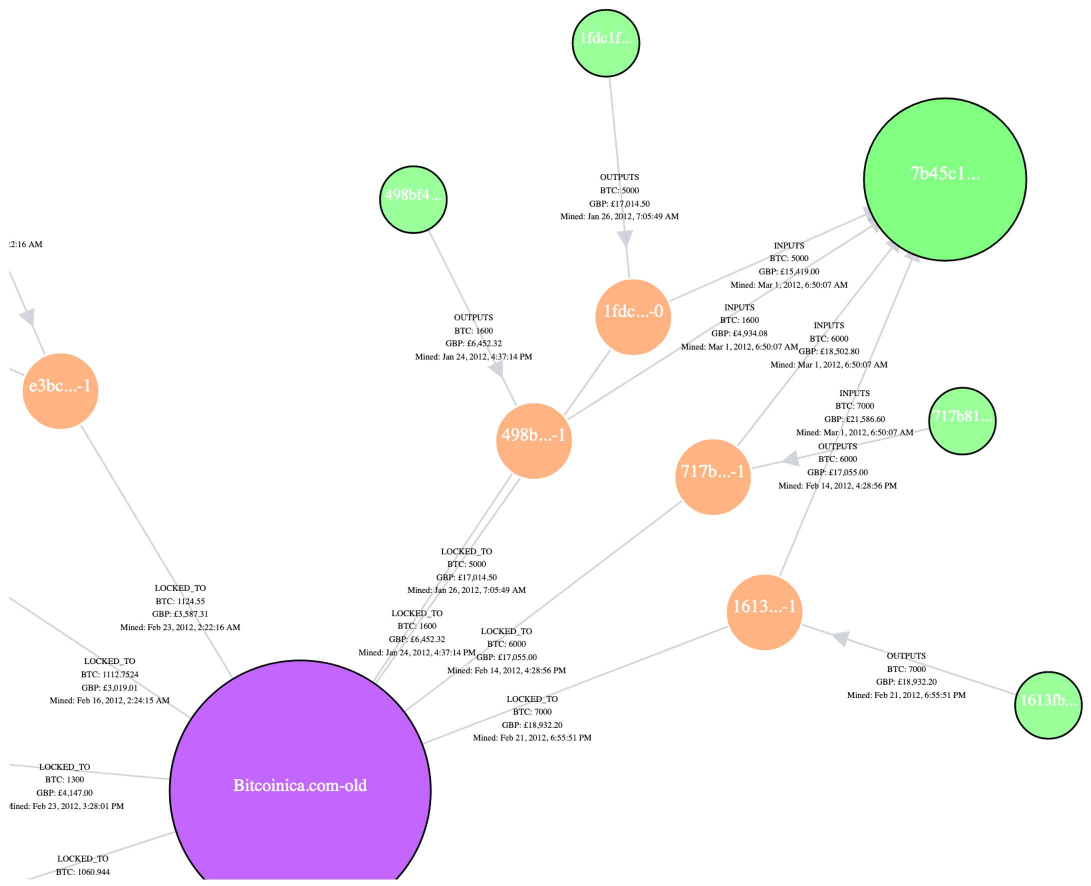

Figure 11.2: A screenshot of the view when searching for entity *Bitcoinica.com*, filtered by date on 1st March 2012 (*note: filtered by the date outputs are spent*) and by price, showing outputs spent with value > 1000BTC. Showing specifically the transaction spending the highest value outputs

.

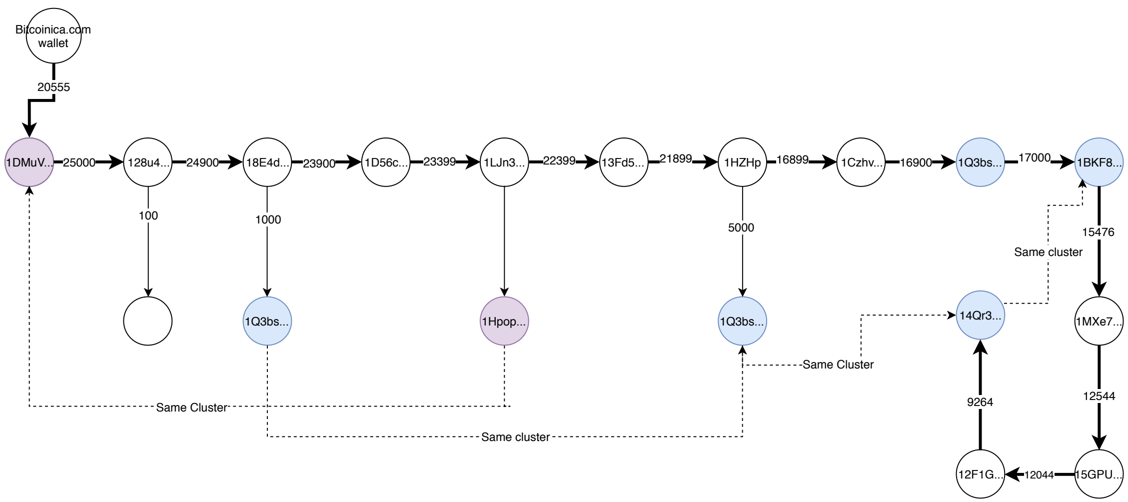

Figure 11.3: A visual representation of the peeling chain discovered using *Radar* while investigating the 1st March 2012 *Bitcoinica.com* theft

.

11.3.1 Areas identified for improvement

While performing the investigation using the tool, there were several features we imagined would have been useful in assisting us with the investigation, but are not currently provided by the tool. These features are:

- Better distinction between incoming and outgoing funds (different colour links?), and potentially the ability to hide/show either incoming or outgoing only.
- Caching search parameters: Many times we would view the result of a search and need to tweak with the filtering parameters, however, the original search data is not cached and needed to be re-entered which was time-consuming.
- Could introduce functionality to automatically identify peeling chains and organise the layout to show the main flows, in a similar way to how the diagram is organised in figure 11.3.

11.4 Path Finding Correctness

To ensure the correctness of the path finding feature, we take Bitcoin addresses that we know have sent funds between eachother as test cases and check that *Radar* returns a valid shortest path between the two addresses. Assuming *Radar* returns a path between the two addresses $a1$ and $a2$ of length n then we can validate that this is in-fact the shortest path by inspecting all other nodes os reachable from $a1$ and $a2$ of length $n-1$. If starting from $a1$ and we find $a2$ in os then we have found a shorter path and the path of length n was not the shortest. Given that n is the shortest path, we then additionally validate that $a2$ is reachable from $a1$ in a path of length n.

One example test used the address `1HB5XMLmzFVj8ALj6mfBsbifRoD4miY36v` and `3AnGAuh5BJ9sJ7TgkZ11XJDPGbaQGs4haM` as the start and end nodes. This returned the result shown in figure 11.4. Repeated validation using this technique led us to be confident that the path finder feature is in-fact correct.

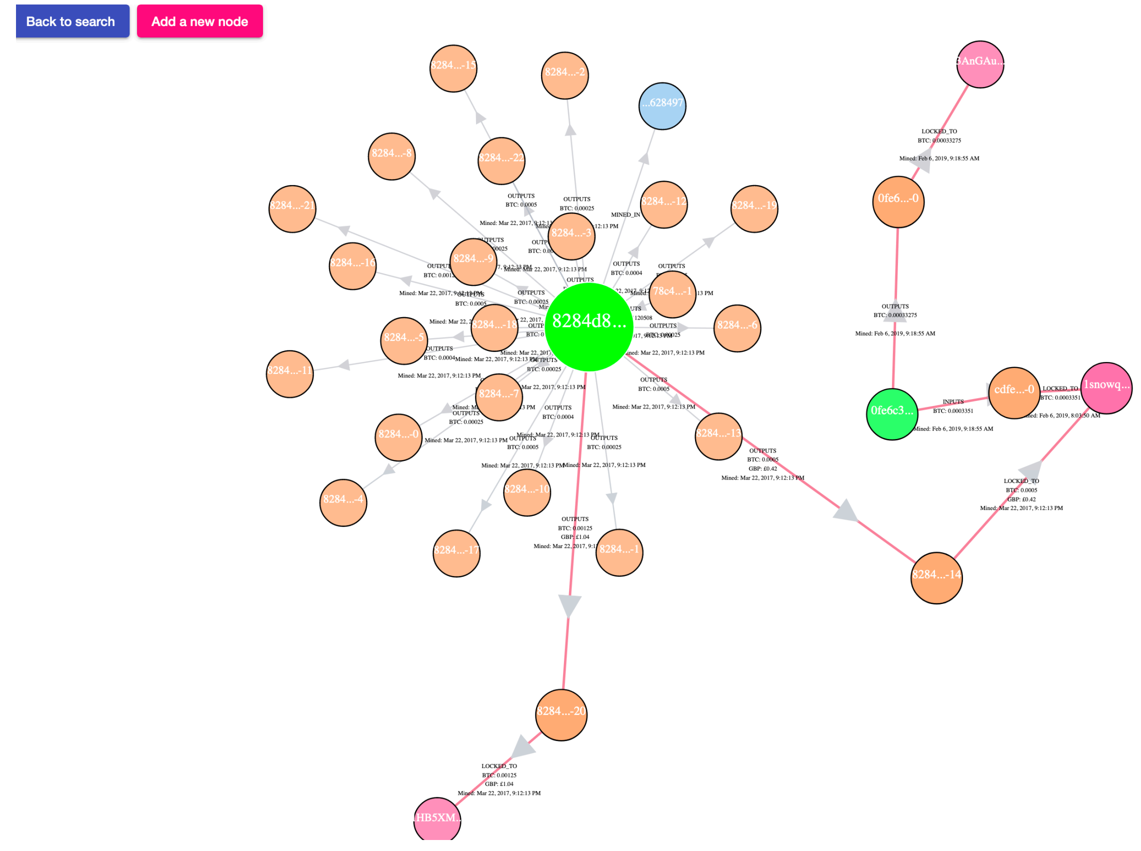

Figure 11.4: A screenshot of *Radar* showing the result of a path finder query between addresses `1HB5XMLmzFVj8ALj6mfBsbifRoD4miY36v` and `3AnGAuh5BJ9sJ7TgkZ11XJDPGbaQGs4haM`.

11.5 Clustering Correctness

We can verify the correctness of *multi-input* clustering by searching for a transaction known to have several inputs locked to distinct addresses. For example, we were able to verify the correctness of clustering with immediate neighbours using address `17jkFTQuYaGssazzqZ6CTHgRVQYRgLmf34`. This address inputs transaction `8897e...`[2], and also has inputs from 9 other distinct addresses. The clustering algorithm then produces a cluster of 10 addresses that are considered to be controlled by the same user.

We now verify the transitive clustering behaviour of the algorithm. To achieve this, we mutated a local Neo4J database containing a subset of the Bitcoin Blockchain (only blocks 0-2000).

We manually located an instance of addresses that can be clustered transitively using this heurisitc. The visual representation of the connections existing between two transitively linked addresses can be seen in figure 11.5. The 3 addresses in this screenshot (in green) are `15oUEZFKAC8E8BTLt1s1jx4fPxumwB3ecr`, `17iqXQcGfjjHSsJK993mu75sPBnLFSEp8F` and `1Js6Nx1822qSjETsnp2kkhQFwmRNRWxgak`. They are linked through two transactions (in blue)

[2] 8897ea9ceaf18a546cdc513b9179bae31a462ee5bf47818eb7ba909082d11777

04dc7...[3] and c7a4b...[4].

We were able to verify that executing the clustering algorithm included all 3 of these addresses in the result, and became confident of the correct transitive behaviour of the implementation of the clustering algorithm.

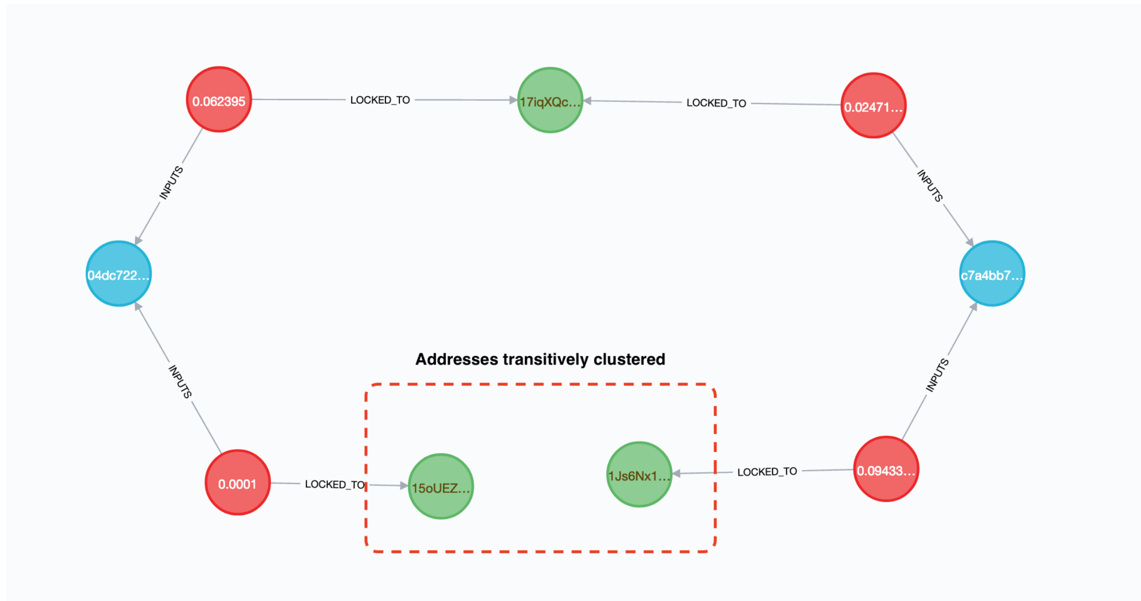

Figure 11.5: A screenshot of the Neo4J UI : Visual representation of addresses that are clustered transitively.

11.6 Missed Objectives

An objective of this project that we, unfortunately, did not have time to complete was applying the same change address clustering in section 8. This was due to the clustering algorithm not completing in the time allotted for this project. Although the feature was not complete, the outline of the algorithm required for implementing this feature can be found in future work section 12.2.6.

11.7 Comparisons with existing tools

As described in the background section 2.7, there currently exist some tools, both free and proprietary, that could be used to help carry out a digital forensic investigation. In this section, we compare the free solutions to *Radar*, both qualitatively and quantitatively.

11.7.1 Wallet Explorer

Given the example above, where we would like to investigate a theft that occurred from *Bitcoinica.com*, we would need to take the following steps on the *Wallet Explorer* site:

[3] 04dc7226529763e8c15d58171d1ac1e28cb6b8d3165005765e65757c7b219d2d
[4] c7a4bb767027a4382462c32b6a824a53e2715d833c0d86e9676a4fbddedca0a9

- Navigate to *walletexplorer.com* **one click**
- Click *bitcoinica.com old* **one click**
- Scan the transactions date table for a date matching 1st March 2012 (the theft date): this required using the table pagination controls until transactions for March 1st 2012 shows : **one click**
- We now search for the large transaction values and see a outgoing transaction of 20,555 BTC and inspect that transaction **one click**. TXID is `7b45c1742ca9f544cccd92d319ef8a5e19b7dcb8742990724c6a9c2f569ae732`
- We see the output address of transaction is `1DMuVKe9PKpx3dbs2b2MnXuVmLfA4drHif` **one click**
- Can click on 'next tx' to follow the flow of funds: pays the funds to address `128u4nNS2DCbPk61aNAXLUTsHZt5FAEtit`
- Can follow the peeling chain in same manner as before with one click on 'next tx' - **n clicks for n steps of the peeling chain**

In comparison, the steps taken to obtain similar data using *Radar* developed in this project are:

- Navigate to *Radar*: **one click**
- **One click** to initiate the search for the *Bitcoinica.com* wallet with all date and price filter applies such that only nodes for 1st March are shown and only transactions of high BTC value are shown
- Can instantly identify the high value transaction `7b45c1742ca9f544cccd92d319ef8a5e19b7dcb8742990724c6a9c2f569ae732` so can trace the funds from that output by double-clicking and seeing the address `128u4nNS2DCbPk61aNAXLUTsHZt5FAEtit` : **one click**
- Can now follow the peeling chain for one click for each step **n clicks for n steps of the peeling chain**

We use a quantitative measure of the 'number of user clicks' to measure the number of steps that need to be taken to follow a peeling chain after a theft for a given wallet.

Using *Wallet Explorer*, we find we require **5 + n clicks** whereas using *Radar*, just **3 + n clicks** when following **n** steps of a peeling chain from the same theft.

Qualitatively, *Radar* made following the peeling chain far easier as you are given the option to see where funds peel off to from any stage - using *Wallet Explorer*, you can only see one step of the peeling chain at any one time; *Radar* shows the entire path and overall paints a better picture of the flow of funds.

Additionally, a factor not considered in the above click metric is if the user wanted to know what the value of each stage was in its respective fiat currency. *Radar* integrates the converted values into the UI; using *Wallet Explorer*, we would have to add several more clicks to measure the steps required to convert the transaction values to a fiat currency.

11.7.2 Blockchain Explorer

We compare the functionality of *Radar* to that provided by *Blockchain Explorer* [see 2.7.1]; we do not use the clicks metric as before due to variations in the functionality provided by each tool. Therefore, we carry out a qualitative comparison.

Blockchain Explorer does not provide the functionality to search by an entity name such as *Bitcoinica.com*; therefore, we would struggle to identify the transactions related to a theft from this entity.

Skipping the step of identifying suspicious transactions and presuming we had a particular transaction of interest, such as `7b45c...`[5]. From this point, we have similar functionality as provided by *Wallet Explorer* and can follow links to see each transaction in isolation and trace the peeling chain, but with the same drawbacks, in comparison to *Radar*, that it does not show the whole path.

A feature that *Blockchain Explorer* does provide is a conversion into USD; however, this is not a historical conversion, this is a conversion according to the current exchange rate. For instance, when viewing the above transaction on the site on 12th June 2019, we see it shows the 20,555 BTC output sent to address `DMuVKe9PKpx3dbs2b2MnXuVmLfA4drHif` has a value in USD of $164,885,221.3. However, when this theft occurred, bitcoin had a value that was magnitudes lower than its value today, therefore this conversion on *Blockchain Explorer* may not be very useful to understanding the value of transactions that occurred during the 2012 theft.

11.7.3 Blockpath

We compare the functionality offered by *Blockpath* [see 2.7.4] to *Radar*. Again, we do not use the clicks metric as before due to large variations in the functionality offered. Therefore, we carry out a qualitative comparison only,

Similar to *Blockchain Explorer*, *Blockpath* provides the ability to search for particular transactions and addresses, but not the ability to find all activity for an entity such as *Bitcoinica.com*, so it would be difficult to use this tool to identify a suspicious address.

A distinguishing feature *Blockpath* provides that the other tools do not is a graphical interface. Using a given suspicious transaction id `7b45c...`, we attempt to follow the peeling chain using the graphical interface. Similar to *Radar*, funds can be followed by double clicking on nodes which will expand the next step in the flow of funds.

A feature of *Blockpath*'s graph visualisation which we believe is very useful is the timeline on the left-hand side [see fig 11.7], and organisation of nodes vertically based on their timestamps. The organisation of nodes in *Blockpath*'s graph visualisation is, therefore, more meaningful than in *Radar*, where the arrangement is random.

However, *Blockpath* does not provide the option to filter such that only transactions for a particular date or value are shown; this, therefore, leads to bloat of the graph which could make tracing the flow of funds more difficult [see figure 11.6].

[5] 7b45c1742ca9f544cccd92d319ef8a5e19b7dcb8742990724c6a9c2f569ae732

Figure 11.6: A screenshot of the *Blockpath* graphical interface feature.

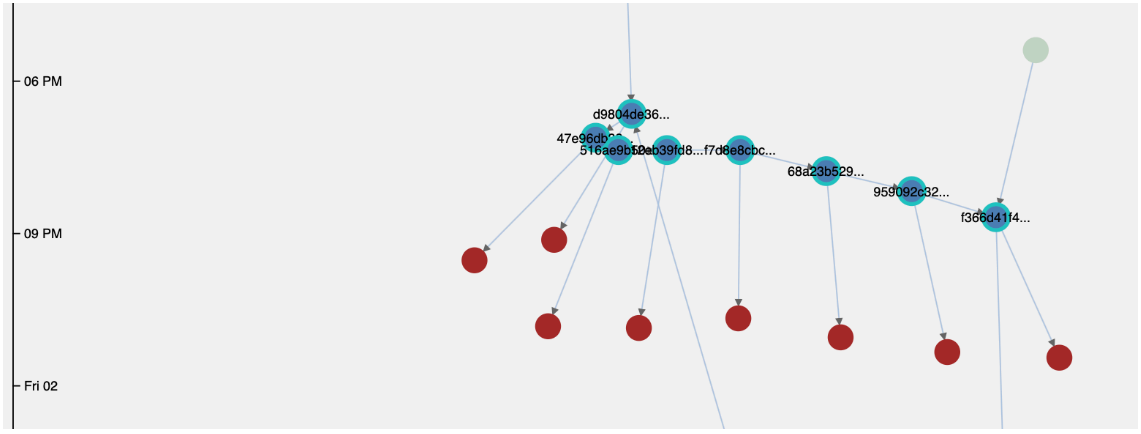

Figure 11.7: A screenshot of the *Blockpath* graphical interface feature, also showing timeline.

11.8 Risks

The greatest element of risk was posed by the import task conducted in section 6; having a functioning graph database populated with historical Bitcoin data was the foundation upon which all other contributions in this project were based on. It was therefore crucial that this deliverable of the project was successful at an early stage.

Another significant element of risk is the overhead of interacting with data in such a vast dataset; without careful consideration of query performance and latencies, much of the functionality would become unusable. Queries could take an infeasible length of term to produce a result or fail completely due to memory limits being exceeding while building a response (such as when finding path that references a huge number of neighbouring nodes).

11.9 Summary

Here we summarise our findings from the evaluation to discuss some of the more significant weaknesses and strengths of this project.

11.9.1 Weaknesses

A significant weakness of this project is that the Bitcoin data accessible through *Radar* is historical only; we did not have time to implement a mechanism for automatically keeping the database up to date with new Bitcoin data. How this would be done is suggested in the Future Work section 12.2.1. This is a significant limitation as the more recent investigations cannot be automatically investigated using *Radar* if they occur beyond block height 570k (3rd April 2019); a possible solution here, for the current implementation, is to use *Astrolabe* to download blocks 570k+ and the process from section 6 to re-populate the database including the new blocks. However, this would require manual intervention and therefore a higher maintenance effort of this product.

Another weakness is that the entity to address associations collected by *Quadrant* in section 5 were only retrieved from a single source, *walletexplorer.com*; ideally, we would prefer not to be dependent on *walletexplorer.com* for this data, as it exposes us to inaccuracies in their dataset or risk of losing our entity tagging data source entirely if *walletexplorer.com* becomes unavailable or stops serving the data.

As shown in section 11.2.2, requests made using the path finding feature massively increases overall request latencies for other users and can even lead to DoS for other users; this is a weakness of the current implementation, and can be addressed with a solution proposed in 12.2.2.

11.9.2 Strengths

Searching for an address with the ability of applying date and price filters works very well and is effective; feedback from industry investigators and our own test investigation [from section 11.3] assures us of the effectiveness of these features.

Loran performs well under load, showing it can handle 100 concurrent users well in section

11.2.2 (when omitting path finding requests).

The overall investigation tool *Radar* provides more comprehensive features for conducting an investigation than existing tools *Wallet Explorer*, *Blockchain Explorer* or *Blockpath* [see section 11.7].

Chapter 12

Conclusion

12.1 Reflection

The purpose and demand for a tool for digital forensic investigations is very apparent; whether it be used for criminal investigations by bodies such as the Metropolitan Police or meeting compliance requirements at exchanges such as Coinbase.

With appropriate hardware (several powerful cores, plenty of memory and several TB of SSD storage), we have shown the task of importing Bitcoins Blockchain into a graph database is very feasible and can be achieved in a very reasonable amount of time, especially if using Neo4J's batch import tool. We have also shown, through our load testing and query profiling evaluation, a graph database is an appropriate storage solution for a dataset the size of Bitcoin's.

We have discovered, in this project, the continuous investment that providing accurate clustering information requires; whether it be actively curating a list of addresses known to be under control of a particular exchange, or by hardening clustering algorithms such as *change-address* by using new Bitcoin data to influence previous clustering decisions and achieve a better true-positive clustering rate.

We found through our several approaches of providing clustering information that address clustering information should be stored with the Blockchain data itself rather than attempting to perform clustering on demand; clustering on-demand can lead to dramatic variations in request response times based on the size of a cluster.

We have shown the effectiveness of a tool, such as *Radar* which provides graphical Blockchain visualisation with granular filtering controls, in enabling suspicious transactions to be easily identified; we further have shown the application of the same-input clustering heuristic in helping with the identification of peeling chain structures.

12.1.1 Future of Crypto-Currency Law Enforcement

The uptake of crypto-currencies with even stronger privacy-preserving features is a cause for concern for law enforcement; cryptocurrencies such as Monero [see 2.10.4] render many current cryptocurrency investigation techniques unsuitable. Mat stated this is an issue, however, due to Monero's relative difficulty of use, a very low proportion of criminals are

thought to be using it, in comparison to Bitcoin usage. There is additionally hurdles in the way of users wishing to buy and sell Monero; Mat referred to an example where, in Japan, exchanges will have their license remove if they accept Monero.

12.1.2 Working with Data at Scale

Data engineering on a vast scale was one of the most significant experiences and lessons taken from this project. At almost every stage in this project, it was essential to consider the algorithmic complexity of our implementation, in addition to considering how data is going to be stored when we can no longer assume datasets can fit in memory. Careful consideration of data storage solutions was required, and therefore consideration of the access latencies each will provide for particular queries (e.g. simple key, value lookups could be satisfied by MongoDB with an index, but traversing the many relationships or path finding would be better suited to a graph database).

Throughout the project, we took a different approach to implementation to account for the scale of the dataset. Examples of this include:

- Using Trie data structure for address to entity matching in section 5.3
- Creating index's in Neo4j and MongoDB to enable fast retrieval of data in section 6.4 and 8.5 respectively
- Paginating data requesting/reading, such as when paginating requests to Neo4J in attempts at identifying and writing clustering relationships using Spring Data or reading huge CSV files in later clustering attempts in section 8.

Even with careful planning and anticipation regarding the size of the dataset, some implementations didn't work out (running into several *OutOfMemory* exceptions and stack-overflows on the way), and an alternative approach was required; the several attempted implementations clustering is an example of this.

Working with data of this scale also frequently required consideration of how to best parallelise work in order to utilise the hardware at our disposal fully. Leveraging all CPU cores by parallelising tasks greatly expedited the time taken for tasks to complete. An example includes the parallelised asynchronous approach to downloading bitcoin data by issuing many concurrent requests to the RPC endpoint in section 3, using many Amazon EC2 instances to concurrently scrape the walletexplorer.com website for entity tagging data or forking several processes to distribute the workload of identifying addresses to cluster in section 8.5.

12.2 Future Work

Through our own vision of the end product of this tool, our discussions with the Metropolitan Police and Coinbase and through process of evaluating *Radar's* performance and ability to investigate historical thefts, we have curated a series of improvements and future directions to take this project with the end goal of building a tool for professional use.

12.2.1 Infrastructure for Keeping Database up to Date

One of the highest priority features for extending this project will be to implement the infrastructure required for keeping the Neo4J database up to date with the latest Bitcoin data.

This will additionally include extending work in section 5 and 4 to collect the latest wallet clustering information and exchange rate information for the new Bitcoin data. This has the goal of having a Neo4J database that reflects the current state of the Bitcoin Blockchain and will keep up to date autonomously, requiring no maintenance or physical intervention to do so.

12.2.2 Path Finding User Experience

As shown in the evaluation [see 11.2.2], several concurrent path finding requests may lead to denial of service for several users. Additionally shown in the evaluation when profiling a single path finding request, section 11.2.1, the performance and user experience still has much to be desired.

We believe a solution to improving the user experience when using the path finding feature could be to transform the feature from an interactive function to an asynchronous one. Path finding across such a huge data-set is a very resource intensive operation; solutions could include further investment in hardware, such as purchasing more physical hardware or delegating path finding to a container-based cloud solution. However, without this investment, there is little that can be done to reduce the demand on resources that path finding requires. Therefore, an alternative solution could be to change path finding into a background job that takes lower precedence to the other interactive tasks such as searching for an address.

Providing asynchronous functionality would consist of enabling the user to submit a query and returning the result to the user at a later date, in a non-interactive way; possibly through a unique link sent via email. The unique link, when followed, will then render the graphical result of their path finding query. In this way, the user does not become frustrated waiting for long path finding queries to complete, and other users are impacted less by other users submitting expensive queries.

12.2.3 Incorporate Information from more Sources

Meeting with the Metropolitan Police and Coinbase led me to discover several new tools and data sources that they often use in their investigations; some provide free API's that could be used to integrate into this project and provide more features in the investigation view. For example, *ShapeShifter* could be used to flag transactions which exchange bitcoin for another currency; possibly providing a feature that allows funds to be tracked across several crypto-currencies. This would be most useful if *Radar* provided support for several more cryptocurrencies in addition to Bitcoin. Additionally, *bitcoinswhoswhos.com* can be used to flag addresses that have been reported as being involved in scams or have tags associated with them.

Integrating this data into the investigation tool will help make this potentially useful data more accessible to an investigator, reducing the number of steps the user will need to take to retrieve it themselves.

Furthermore, as discussed in the evaluation, the entity to address associations are made based on data from a single source (walletexplorer.com). An improvement would be to retrieve this data from several sources in order to augment the existing entity to address mapping dataset, in addition to confirming additional mappings. Confirming mappings from multi-

ple data sources could provide information to build a 'clustering confidence' rating, which could provide a risk/compliance score for how likely an address genuinely belongs to a cluster.

Address tag data could also be sourced ourselves, by taking a Similar approach to Meiklejohn et al. [18], where addresses are associated with services and exchanges by interacting with the services and exchanges. Obviously, this approach will not be free of cost since real transactions will need to be made, so it is potentially not feasible to do on a rolling basis.

12.2.4 Saving Investigations

Radar currently provides no functionality to save or store the state of the investigation view. When performing an investigation, it would rarely be an instantaneous process; an investigation would often span over a period of time such as several days or weeks. This assumption was confirmed through our discussions with the Metropolitan Police. Therefore, an important feature would facilitate the user to initiate several investigations and save their current state for re-visiting at a later date.

This functionality also highlights the importance of the deployment plan if this tool were to be used in a professional environment; it would be necessary to consider how investigation data is stored securely such that it cannot be accessed by unauthorised users or the owners of the product (us). This would potentially require an on-site hardware deployment for on-site (the user's location) data storage, or security guarantees regarding the isolation and accessibility of information if stored off-site (our location/cloud provider).

Further functionality related to investigation state, which was highlighted by the Mat Stanley as a useful feature, would be to identify when an address re-appears in an investigation that has appeared in previous investigations by that user/group of users.

12.2.5 Exporting Investigation Data

Data from an investigation conducted using *Radar* may be required as evidence, such as to present to a jury in court. Therefore, the data must be exportable as a standardised format, such as CSV. It would additionally be useful to be able to share an investigation with colleagues; a useful feature would be the facility to send a unique link to another user who generates the same investigation view.

12.2.6 Change Address Clustering Heuristic

The first useful clustering heuristic to add to *Radar* will be the *change-address* heuristic as described in section 2.3.2 with infrastructure in place to continuously cluster new incoming bitcoin data and improve existing clustering on historical data (i.e. remove addresses previously considered change-addresses if they are used again as outputs of new transactions).
Algorithm:

- for each transaction tx on the blockchain:
    - get outputs $outs$ of tx
    - get inputs ins of tx
    - initialise valid change coutner = 0

  - continue if length of outputs < 2
  - for each output out of $outs$:
    * fetch address a output is locked to
    * check a has no other outputs locked to it (robustness: change address has one input), continue otherwise
    * check out is not in ins (robustness: not self-change), continue otherwise
    * out is a valid change output, increment valid change counter, remember a as $validChangeAddress$
  - iff valid change counter $==1$, return $validChangeAddress$ as a valid change address

- If there was a valid change address $validChangeAddress$, then create new relationship between addresses which input the transaciton and the change address

12.2.7 More Clustering Heuristics

Additional heuristics to incorporate into *Radar* could include behaviour based analysis, as described in section 2.3.5 or consumer heuristic and optimal change heuristic as discussed in section 2.3.3 and 2.3.4 respectively. For every clustering heuristic that id added, it should be configurable by the user (i.e ability to turn any combination of clustering heuristics on/off) as each may provide the user with different information.

Another method of grouping/tagging distinct addresses could be by the address type. As discussed in section 2.1.2 there are different types of Bitcoin addresses, such as Pay-to-PubKey-Hash (P2PKH) addresses or Pay-to-Script-Hash (P2SH) addresses. It may be useful to make a visual distinction between these types of addresses as they could potentially represent a different role in an organisation; for instance, a P2SH address used in a multi-signature transaction may be useful in understanding what the purpose of the transaction is, compared to if it was a P2PKH address.

12.2.8 Set up Watches for Nodes

Rather than periodically checking if a particular address has spent its outputs, it would be far more useful and efficient for the investigator to be able to set up a notification whenever some particular state regarding that address changes. This idea could be extended to entire wallets, such as watching transactions exceeded a certain value being spent by the 'Mt.Gox' wallet. Additionally, it could perhaps be possible to create watches for structural/usage patterns occurring across the Blockchain. For example, 'watch for this address being involved in a peeling chain'.

12.2.9 UI Improvements

Some smaller improvements/feature additions to the UI that can improve existing functionality

- Add a UI feature that acts as a legend for the different types of nodes
- Extend path finding tool to also accept entity names in addition to addresses
- Ability to collapse nodes relationships and remove nodes entirely

- Auto-populate search form fields with previous search data
- Improvements to price & date filtering by allowing 'greater than date/value' or 'less than date/value' rather than requiring a range be provided
- Improve the organisation of nodes: more intelligent layout of nodes based on their type, the direction of flow of funds, the volume of the funds etc.

12.2.10 Several Crypto-Currencies

Several of the existing digital forensic tools [see 2.7] support several additional crypto-currencies to Bitcoin, including Ethereum, Litecoin, Bitcoin Cash and Dash. Providing support for these currencies would be a natural extension of this project; however, it will be no small task. Infrastructure will be required to perform a bulk import of all of the currencies respective blockchains and infrastructure able to keep all blockchains up to date with the most recent data. In addition, the various clustering heuristics will likely differ per Blockchain, so could need re-implementation or different heuristics provided all-together for each distinct chain.

A potential constraint here could be hardware; the download of several blockchains will require a very sizeable amount of SSD (for reasonable access latency) which may require considerable financial investment.

Bibliography

[1] Sean Foley, Jonathan R. Karlsen, and Tlis J. Putni. Sex, drugs, and bitcoin: How much illegal activity is financed through cryptocurrencies? *The Review of Financial Studies*, 32 (5):1798–1853, 2019. pages i, 2

[2] Fergal Reid and Martin Harrigan. pages 197–223. Security and privacy in social networks. Springer, 2013. pages 1

[3] Cambridge Judge Business School. Distribution of leading cryptocurrencies from 2015 to 2018, by market capitalization, March 2019. URL https://www.statista.com/statistics/730782/cryptocurrencies-market-capitalization/. Date Accessed: June 10, 2019. pages 1

[4] Omkar Godbole. Bitcoin price primed to test 20k usd ahead of cme launch, December 17 2017. URL https://www.coindesk.com/sell-news-bitcoin-price-tests-20k-ahead-cmes-futures-launch. Date Accessed: June 8, 2019. pages 2

[5] Rainer Bhme, Nicolas Christin, Benjamin Edelman, and Tyler Moore. Bitcoin: Economics, technology, and governance. *Journal of Economic Perspectives*, 29(2):213–238, 2015. pages 2

[6] Adam Jezard. How bad is bitcoin crime?, April 8 2018. URL https://www.worldgovernmentsummit.org/observer/articles/how-bad-is-bitcoin-crime. Date Accessed: June 8, 2019. pages 2

[7] Hiroki Kuzuno and Christian Karam. Blockchain explorer: An analytical process and investigation environment for bitcoin. pages 9–16. IEEE, Apr 2017. doi: 10.1109/ECRIME.2017.7945049. URL https://ieeexplore.ieee.org/document/7945049. pages 2

[8] Blockchain.com. Total number of transactions. URL https://www.blockchain.com/charts/n-transactions-total. Date Accessed: June 8, 2019. pages 3

[9] Andreas Antonopoulos. *Mastering Bitcoin, 2nd Edition*. O'Reilly Media, Inc, 2 edition, Jun 21, 2017. ISBN 9781491954386. URL http://proquestcombo.safaribooksonline.com/9781491954379. pages 4, 5, 7, 8, 9, 10

[10] Blockchain.com. Hash rate, January 2019. URL https://www.blockchain.com/charts/hash-rate?timespan=all. Date Accessed: 13 January, 2019. pages 7

[11] Blockchain.com. Blockchain difficulty, January 2019. URL https://www.blockchain.com/charts/difficulty. Date Accessed: 13 January, 2019. pages 7

[12] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008. URL `http://www.bitcoin.org/bitcoin.pdf`. pages 8

[13] Malte Moser. Anonymity of bitcoin transactions an analysis of mixing services. 2013. pages 10

[14] M. Moser, R. Bhme, and D. Breuker. An inquiry into money laundering tools in the bitcoin ecosystem. In *2013 APWG eCrime Researchers Summit*, pages 1–14, Sep. 2013. doi: 10.1109/eCRS.2013.6805780. pages 10

[15] Thibault de Balthasar and Julio Hernandez-Castro. An analysis of bitcoin laundry services. 09 2017. doi: 10.1007/978-3-319-70290-2_18. pages 11

[16] M. G. Reed, P. F. Syverson, and D. M. Goldschlag. Anonymous connections and onion routing. *IEEE Journal on Selected Areas in Communications*, 16(4):482–494, May 1998. doi: 10.1109/49.668972. pages 12

[17] Elli Androulaki, Ghassan O. Karame, Marc Roeschlin, Tobias Scherer, and Srdjan Capkun. Evaluating user privacy in bitcoin. In *Financial Cryptography and Data Security*, pages 34–51. Springer, 2013. URL `http://book.itep.ru/depository/bitcoin/User_privacy_in_bitcoin.pdf`. pages 12

[18] Sarah Meiklejohn, Marjori Pomarole, Grant Jordan, Kirill Levchenko, Damon McCoy, Geoffrey M Voelker, and Stefan Savage. A fistful of bitcoins: characterizing payments among men with no names. In *Proceedings of the 2013 conference on Internet measurement conference*, pages 127–140. ACM, 2013. pages 13, 14, 15, 45, 99

[19] Jonas David Nick. Data-driven de-anonymization inbitcoin. Technical report, ETH Zurich, August 9, 2015. pages 13, 14

[20] John V. Monaco. Identifying bitcoin users by transaction behavior. volume 9457, page 15, 2015. URL `https://doi.org/10.1117/12.2177039`. pages 14

[21] D. Ermilov, M. Panov, and Y. Yanovich. Automatic bitcoin address clustering. In *2017 16th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pages 461–466, Dec 2017. doi: 10.1109/ICMLA.2017.0-118. pages 14

[22] Bitcoin virtual currency: Intelligence unique features present distinct challenges for deterring illicit activity. Technical report, 24 April 2012. pages 15

[23] Nicolas Christin. Traveling the silk road: A measurement analysis of a large anonymous online marketplace. In *Proceedings of the 22nd World Wide Web Conference (WWW'13)*, pages 213–224, Rio de Janeiro, Brazil, May 2013. URL `https://www.andrew.cmu.edu/user/nicolasc/publications/Christin-WWW13.pdf`. pages 15, 16

[24] U.S. Attorney's Office. Operator of silk road 2.0 website charged in manhattan federal court, November 6 2014. URL `https://www.fbi.gov/contact-us/field-offices/newyork/news/press-releases/operator-of-silk-road-2.0-website-charged-in-manhattan-federal-court`. Date Accessed: January 25, 2019. pages 16

[25] Tom Schoenberg and Matt Robinson. Bitcoin atms are flying under the regulatory radar, 14 December 2018. URL `https://www.bloomberg.com/feature/2018-bitcoin-atm-money-laundering/`. Date Accessed: January 25, 2019. pages 16

[26] Greg Walker. How to import the bitcoin blockchain into neo4j, January 9 2018. URL https://neo4j.com/blog/import-bitcoin-blockchain-neo4j/. Date Accessed: March 25, 2019. pages 20, 82

[27] Cesar Pantoja. How to load bitcoin into neo4j in one day, February 22 2019. URL https://medium.com/tokenanalyst/how-to-load-bitcoin-into-neo4j-in-one-day-b555219ed9d2. Date Accessed: March 25, 2019. pages 21, 26, 27, 82

[28] Max Baylis. Blockchain data analytics and health monitoring. Technical report, Imperial College London, September 7 2018. pages 21

[29] Stephane Traumat. Github: blockchain2graph, 28 Nov 2018. URL https://github.com/straumat/blockchain2graph. Date Accessed: 6 February, 2019. pages 21, 82

[30] CoinDesk. Bitcoin price index api. URL https://www.coindesk.com/api. Date Accessed: April 9, 2019. pages 28

[31] Walletexplorer.com: smart bitcoin block explorer, . URL https://www.walletexplorer.com/. Date Accessed: January 25, 2019. pages 30

[32] Neo4J. Neo4j: What is a graph database and property graph, . URL https://neo4j.com/developer/graph-database/. Date Accessed: March 25, 2019. pages 33

[33] Neo4J. Use the import tool, . URL https://neo4j.com/docs/operations-manual/current/tutorial/import-tool/. Date Accessed: January 19, 2019. pages 34

[34] Michael Hunger and Mark Needham. Effective bulk data import into neo4j, 2016. URL https://neo4j.com/blog/bulk-data-import-neo4j-3-0/. Date Accessed: Feburary 18, 2019. pages 34

[35] Bitcoin blockchain size 2010-2019 statistic, . URL https://www.statista.com/statistics/647523/worldwide-bitcoin-blockchain-size/. Date Accessed: Feburary 18, 2019. pages 34

Appendices

Appendix A

User Guide : *Radar*

A.1 Search

- **Search for an address**: Input the full Bitcoin address into the *Address or Entity* input field in the form *Begin investigation with an address*. If the address exists, you'll be navigated to the **investigation view**.
  - **View transaction values in fiat alternative currencies**: Click the bar labelled *Select a fiat currency* under *Additional search options* to open the expanding filter panel. Select the desired fiat currency.
  - **Apply a date filter**: Click the bar labelled *Filter by date/time* under *Additional search options* to open the expanding filter panel. Enable the filter selecting the toggle button which will show the *Enabled* state. Select the start date of your desired date range by selecting *Start Date* (this can be done using the calendar picker or manually in a **mm/dd/yyyy** format). Do the same for the end date of your desired range. The time defaults to 00:00 of your selected date; drag the slider to change this, if required.
  - **Filter by price**: Click the bar labelled *Filter by price* under *Additional search options*
  - **Limit neighbour rendering**: This will be enabled by default; it helps prevent too many graph nodes being rendered on the UI to prevent performance degrading. However, this will not be a complete result for your search. You can adjust the limit (which corresponds to the number of neighbour nodes shown, of each relationship type, for each node) by selecting *Limit Neighbour Rendering* and then dragging the slider to a value of your choice (range of 0-100). If you wish to disable this feature completely, select the *Enabled* toggle button, such that it turns to *Disabled*.
  - **Enable Input Clustering**: Select the *Input Clustering* bar under *Additional search options* to open the expanding panel. Toggle on by selecting the *Disabled* toggle button so that it changes to *Enabled*.
- **Search for an entity**: Select the *Search Type* field drop-down in the *Begin investigation with an address* and select *Entity Name* from the dropdown. Then enter the full entity name in the *Address or Entity Field*. Filters can be applied in the same way as they are for an address search. If the entity exists, you'll be navigated to the **investigation view**.
- **Search for shortest path between two addresses**: Enter both of the full addresses in the fields *Start Address* and *Destination Address* in the *Path Finder* form. You will receive

a response *No path found between addresses* if no path exists; if a path does exist, you'll be navigated to the **investigation view**.

A.2 Investigate

- **View a node's details**: Hover over any node; this will lead to the node expanding in size and showing a node information panel
- **Copy node data to clipboard**: Simply click on any field in the node information panel, data will be automatically copied to your clipboard.
- **Expand a node's neighbours**: Double click on any node with a black border: this will lead to the node pulsating in size until all neighbours are fetched. The neighbours will be automatically added to the graph.
- **Arrange layout**: Simply drag and drop any node to your desired position
- **Perform a new search**: Click on the blue *Back to Search* button in the top left-hand corner to be navigated to the **search view**
- **Add custom nodes**: Click the pink *Add a new node* button in the top left hand corner. Select the *Custom Node Type* dropdown to select the most appropriate node type. Give the custom node a name by entering its name in the *Name of node* field
  - If *Photographic ID* selected, you'll be able to upload an image: select *Choose File*, select the image from your local file system and select *Upload*
  - You'll be able to add an arbitrary number of fields: select *Add a property* and give each of your properties a unique name in the *Property name* field, then its corresponding value in the *Property value* field
- **Introduce a link between a custom node and any other node**: Hover over a custom node until the custom node view shows. Select the *Create a link* button. Type (or select from auto-complete) the ID of the node to connect to. The ID must be a valid ID of another node currently present on the graph. The link can be given any label by entering the label in the *New Link Label* field. The direction of the link can be reversed clicking the arrow icon.

Appendix B

Radar Views

Below are all of the various views that *Radar* has.

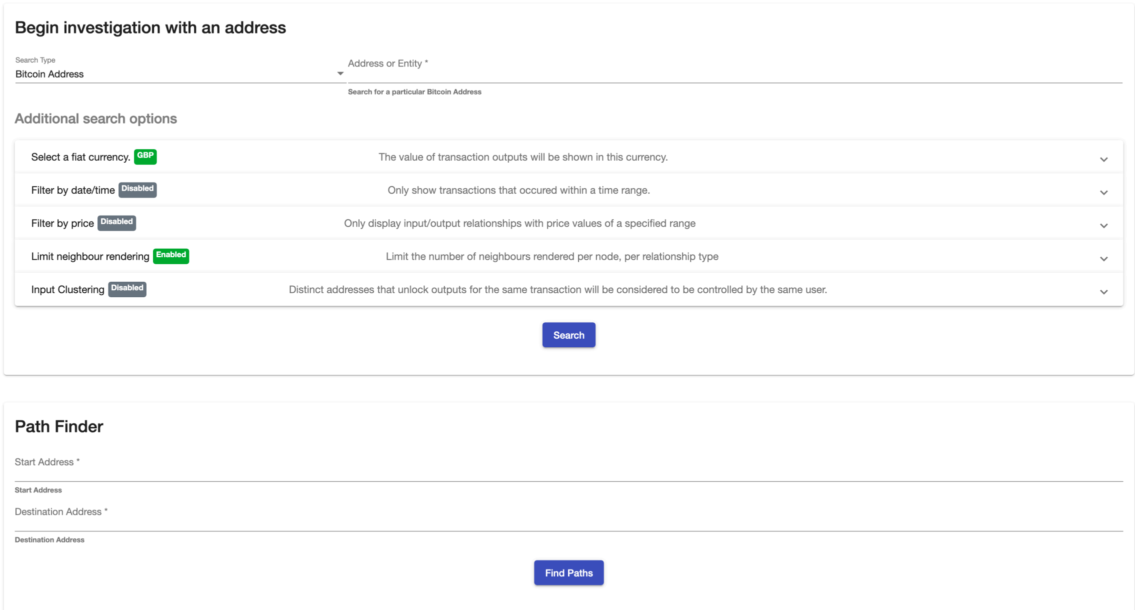

Figure B.1: *Radar's* Search View

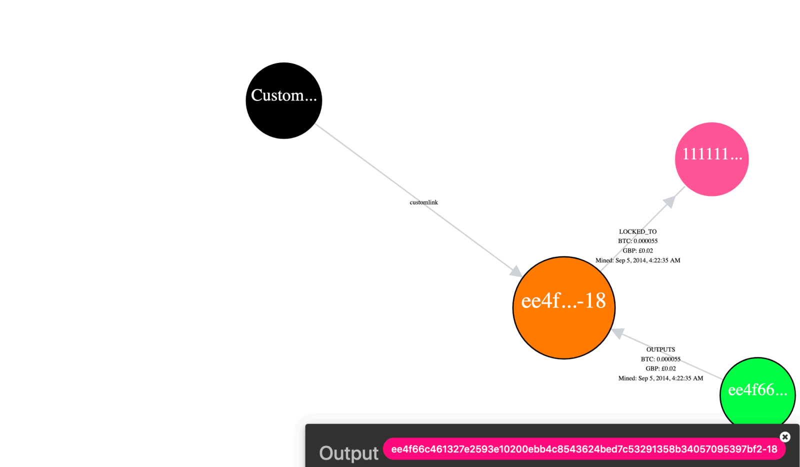

Figure B.2: *Radar's* Investigation View

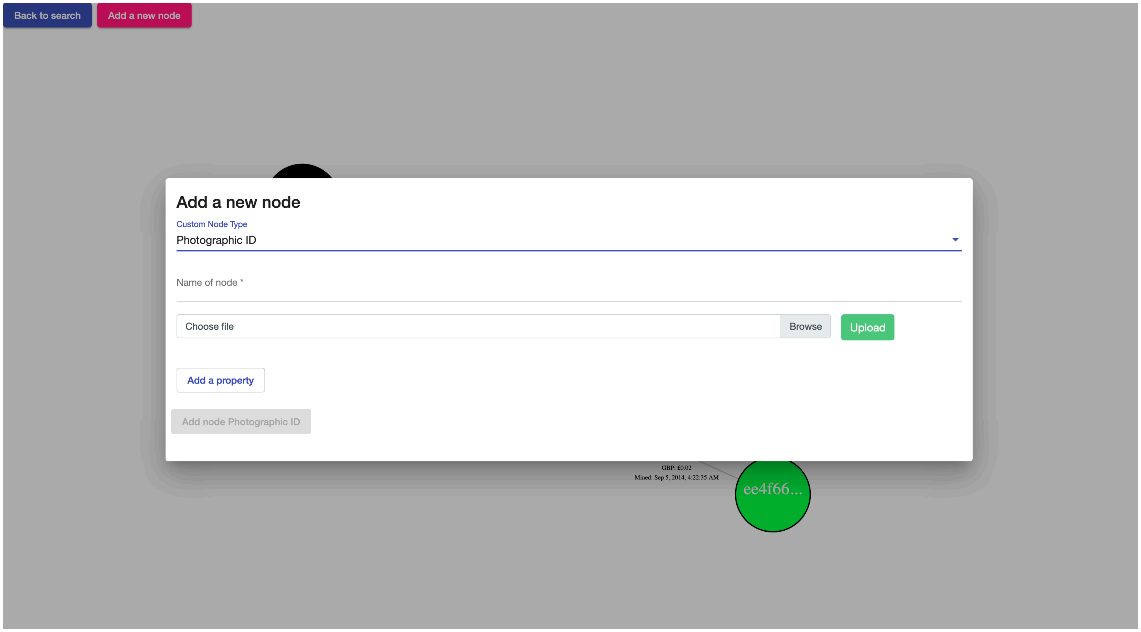

Figure B.3: *Radar's* Investigation View with the *Add Node* overlay displayed

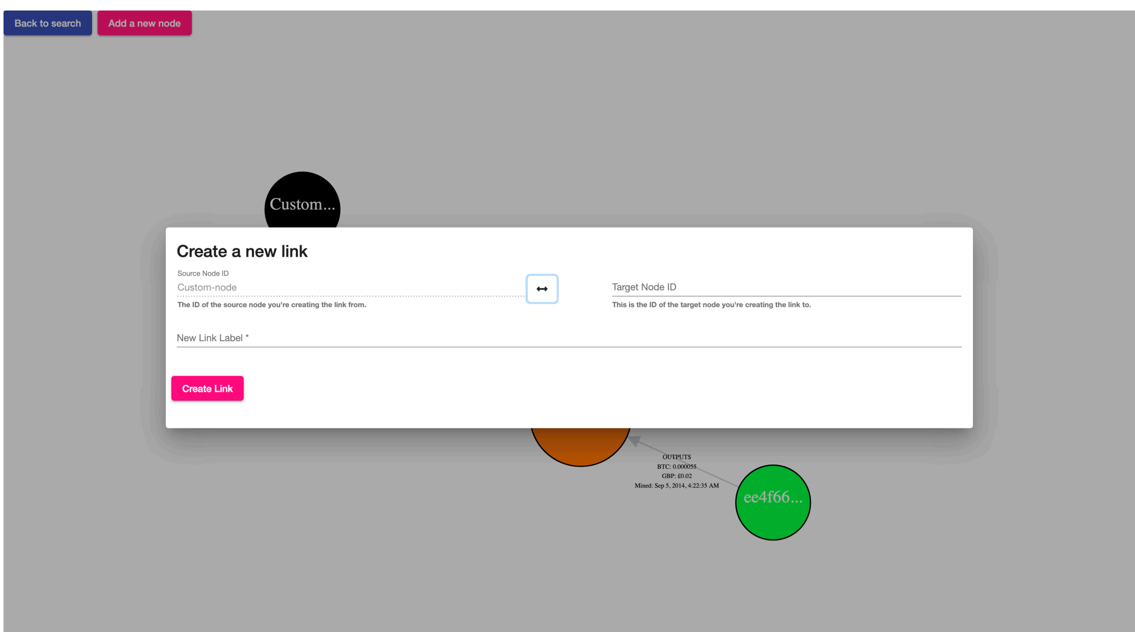

Figure B.4: *Radar's* Investigation View with the *Add Link* overlay displayed

Appendix C

Locust Evaluation Results

Type	Name	# Requests	# Fails	Median (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)
GET	/bitcoin/shortestPath/11111116Jvg5YivHHTcuapzk5CtSEBVA/1111111111111111111AMXmxkqDAw	1	0	523000	523049	523048.65550994873	523048.65550994873	28441
GET	/bitcoin/shortestPath/11111osx23iVHGzuWdFrbcNAUXcACyE/111111111FKoXLpnhTRVqzBrS4adb3	1	0	364000	363774	363774.454832077	363774.454832077	29977
GET	/bitcoin/shortestPath/111111ECWxYc6uAqpiPoyTtv4zHY8u8G/11114Zusqcz7pqcaUP7kbw6bisj4FJXj	1	0	262000	261762	261762.35175132751	261762.35175132751	23650
GET	/bitcoin/shortestPath/111111156CCmffbHaGjy5ufWM9CEGn5V/111111111FKoXLpmfSvPURYxzmc8KU	1	0	257000	257217	257217.19479560852	257217.19479560852	27536
GET	/bitcoin/shortestPath/1111111118AQGAuubX9PafPrRGGEg6/11114zpxK61ZkiLNaMpnMJYEq1MpffVYj	1	0	252000	252418	252418.36595535278	252418.36595535278	26122
GET	/bitcoin/shortestPath/111111ECWxYc6uAqpiPoyTtv4zHY8u8G/11111197ivEXesq2WkWbVkZPg4kdLDuy	1	0	217000	217197	217197.04484939575	217197.04484939575	21228

Figure C.1: The Locust statistics dashboard for the requests ordered by decreasing response times; showing the shortest path requests to be responsible for response time spikes

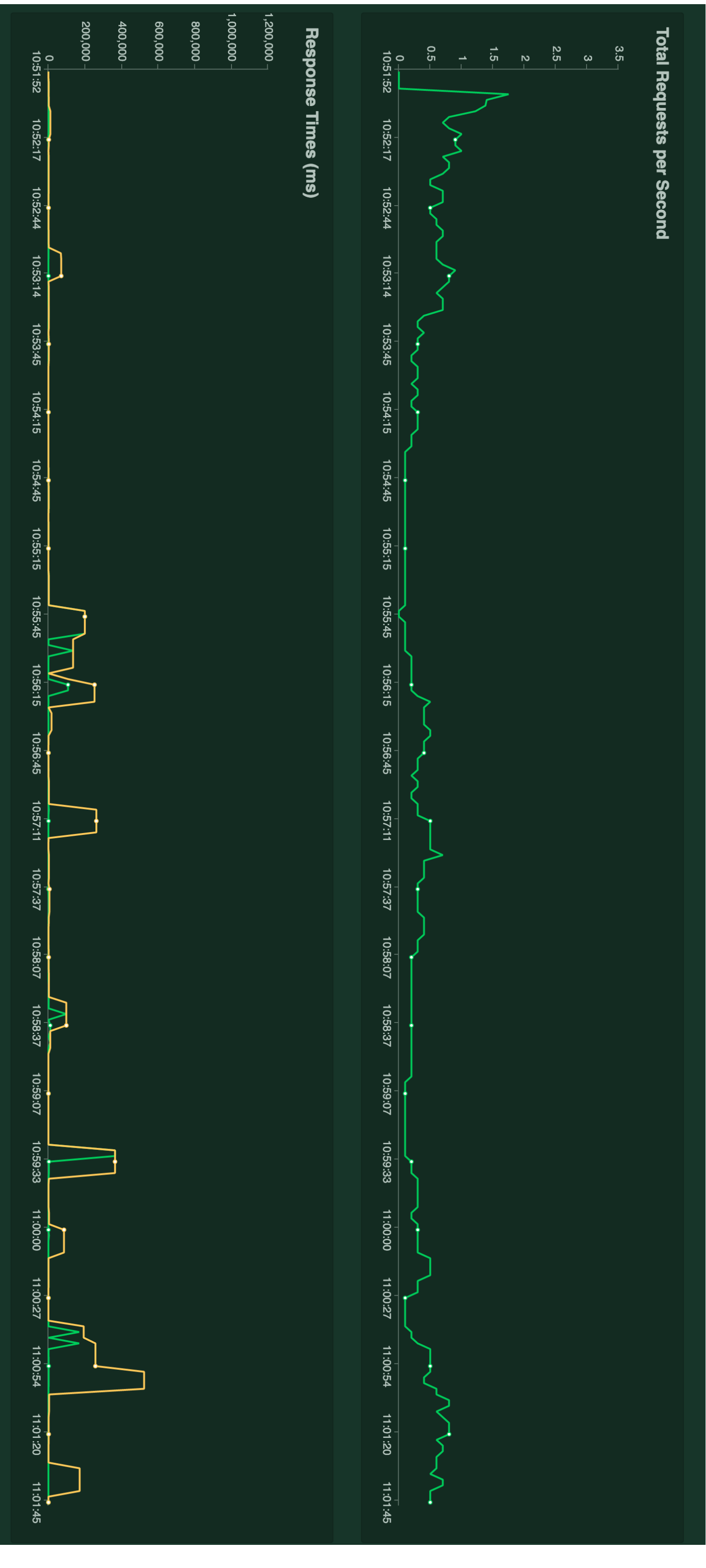

Figure C.2: Locust Statistics Dashboard for load of size 10 users

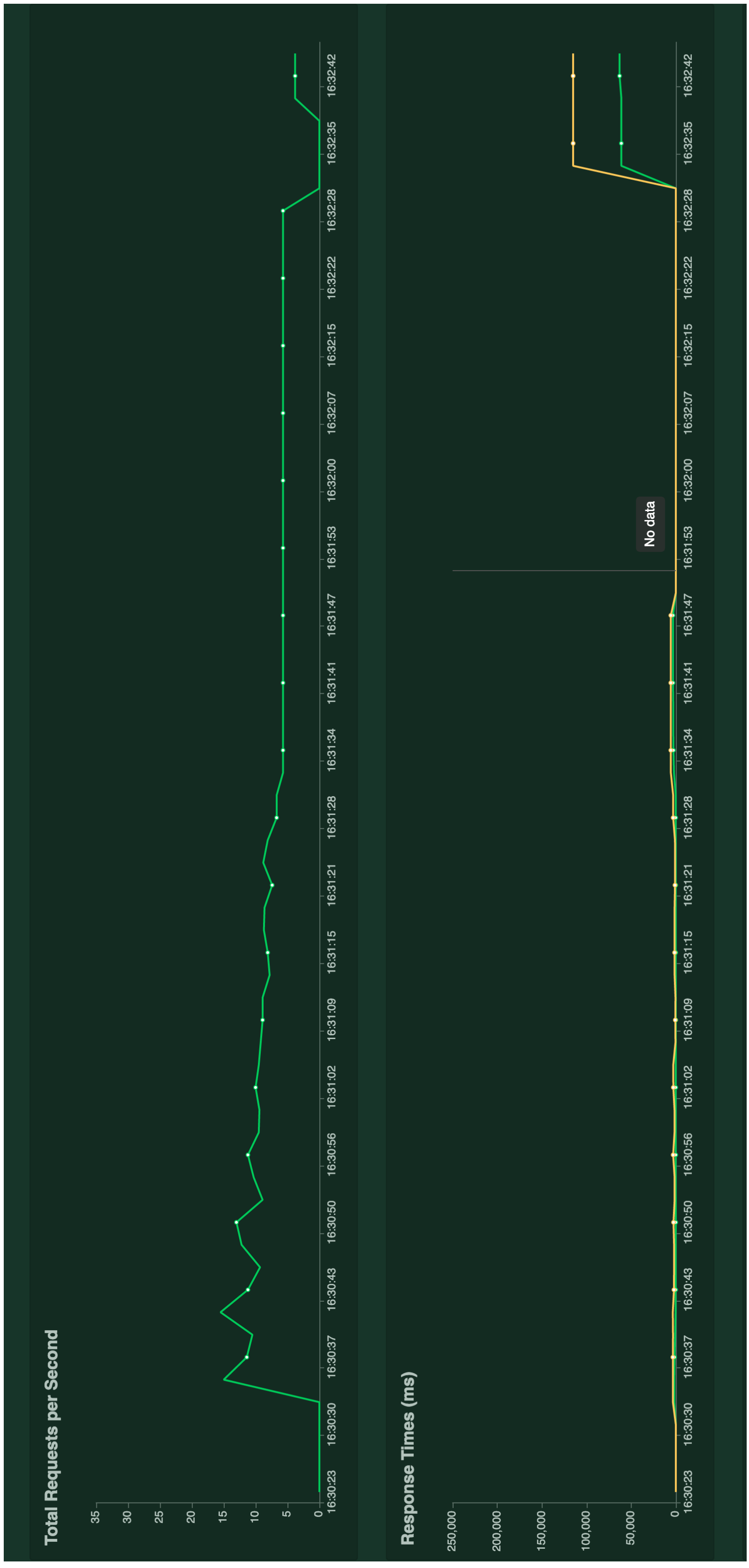

Figure C.3: Locust Statistics Dashboard for load of size 100 users

Type	Name	# Requests	# Fails	Median (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)
GET	/bitcoin/shortestPath /111113EWFMcLpRD2 25HuYjeHab4tJKw1H /111111119cyhpnE4ouZ 9p5FRsVR1evz	1	0	123000	123081	123080.78622817993	123080.78622817993	26412
GET	/bitcoin/shortestPath /11111111111t111111111 11Vq1Wm3 /11112FmfChN4cYTVg kYweC2waCqxBB1VX	1	0	115000	114779	114779.25753593445	114779.25753593445	31794
GET	/bitcoin/shortestPath /1111F8tE8v1Y1twLmJ 8i5ck9JdSFGYhn1 /1111111118AQGAxiLT ET93q82NcCji	1	0	112000	112185	112184.52000617981	112184.52000617981	20533
GET	/bitcoin/shortestPath /111111811zDiVJNrGou fFmZUdetqXMqV /1111122sz8wuwHFwU CkJv2mks42zVk2DD	1	0	109000	108653	108652.54235267639	108652.54235267639	16599
GET	/bitcoin/shortestPath /11114QYVbLosJTVq6 EMY475nzM8o5EBCp /1111111111111h111111 111WdQPXP	1	0	108000	108210	108210.46304702759	108210.46304702759	31810
GET	/bitcoin/shortestPath /111111QS7uV522Ase VV6Jqs7ZTb5Ewa6 /11114Jyn5pfYhN16KX cFKAAmWWd2QvxKP	1	0	106000	105627	105626.66606903076	105626.66606903076	31108
GET	/bitcoin/shortestPath /11111115AXAXTGfP6g GfRMpCQy5xSTg4 /1111111111111o111111 111cLV3wA	1	0	101000	100980	100980.4961681366	100980.4961681366	31780

Figure C.4: The Locust statistics dashboard for the requests ordered by decreasing response times; showing the shortest path requests to be responsible for response time spikes

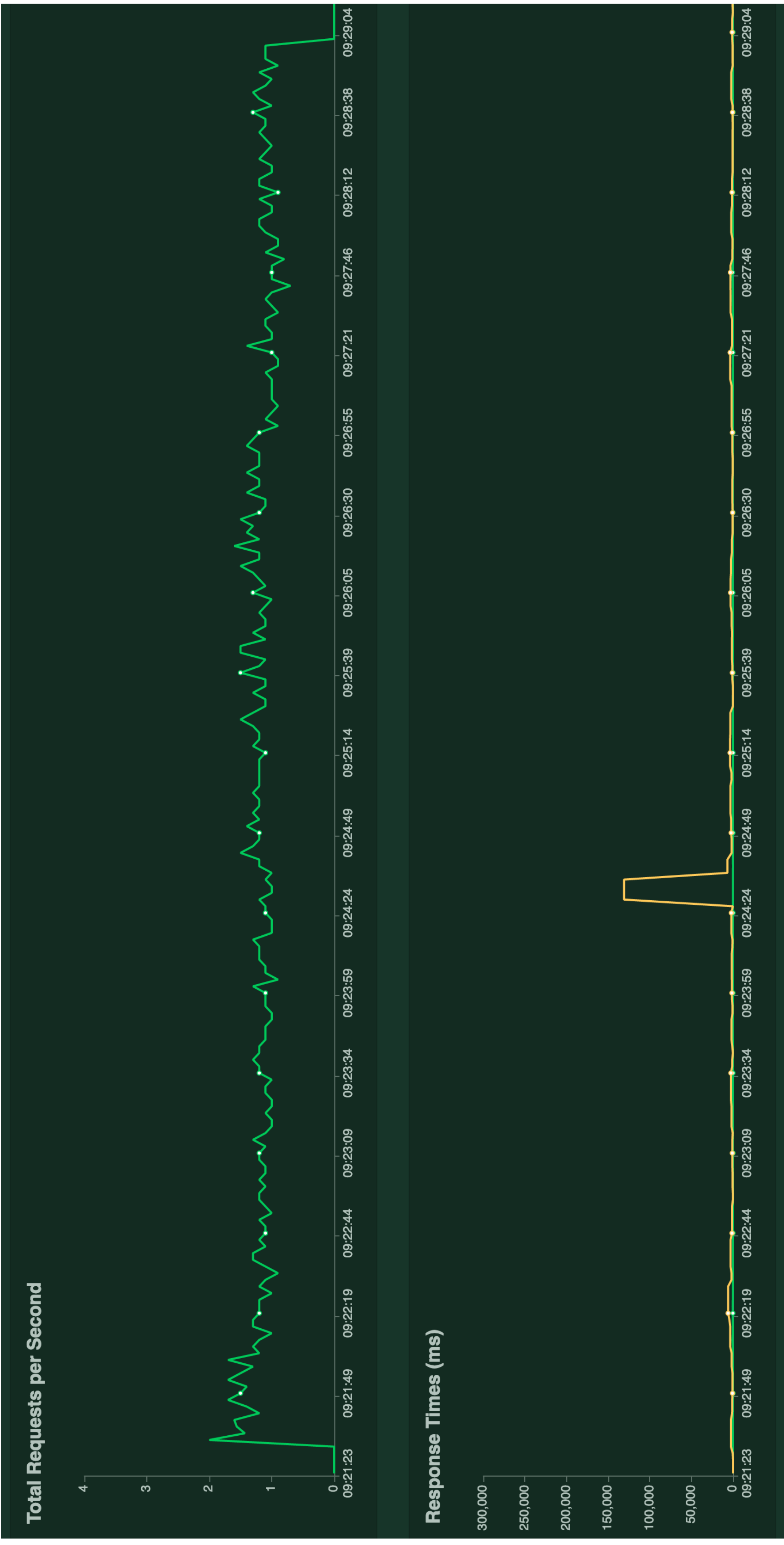

Figure C.5: Locust statistics dashboard for load of size 10 users, without shortest path requests

Type	Name	# Requests	# Fails	Median (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)
GET	/bitcoin/output /80615a7ae1e1e1771f ce0aed437f76b5d88cf 93314c2a99093fb1cb8 ec93b35e-1	1	0	87000	87158	87158.17523002625	87158.17523002625	31378
GET	/bitcoin/output /41da43172d021dda90 4775141e07d61fd5b3b 57ed89cc35a473f4fa5 a1ea25ca-1	1	0	27000	27348	27347.520112991333	27347.520112991333	8928
GET	/bitcoin/output /021fa6992d318236c5 3aa101402fe8a6606f4 7d1047c1a778601b9cc 3c600f96-2	1	0	20000	20318	20317.86060333252	20317.86060333252	6769
GET	/bitcoin/output /80615a7ae1e1e1771f ce0aed437f76b5d88cf 93314c2a99093fb1cb8 ec93b35e-4	1	0	18000	18100	18100.335836410522	18100.335836410522	15011
GET	/bitcoin/output /1ad0cde3e65643a016 109c4247eebf3a45408 ed40d2dfe3c380517bc fbd1754a-1	1	0	18000	17546	17546.46372795105	17546.46372795105	12464
GET	/bitcoin/output /d09588f509c5bc69115 0263487ae685371bcd 0785f1c5ae0de84990d f504f186-1	1	0	17000	16996	16995.58186531067	16995.58186531067	8551
GET	/bitcoin/output /65757681e5cf56742fe e8bd7dc3cfd32a5013d 8d711f0d281d2c539f4 5f0188c-1	3	0	640	5349	629.5380592346191	14779.496431350708	5404

Figure C.6: Locust statistics dashboard for load of size 10 users, showing requests ordered by decreasing response times; showing the output requests responsible for the spikes in figure C.5

.

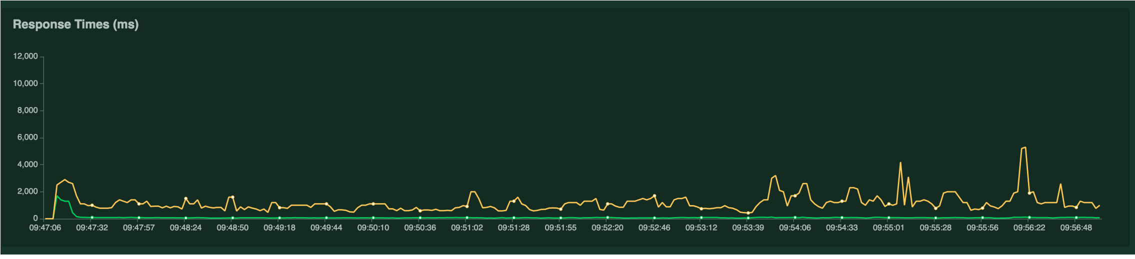

Figure C.7: Locust statistics dashboard for load of size 10 users, without path finding requests and including a node limit of 10.

.

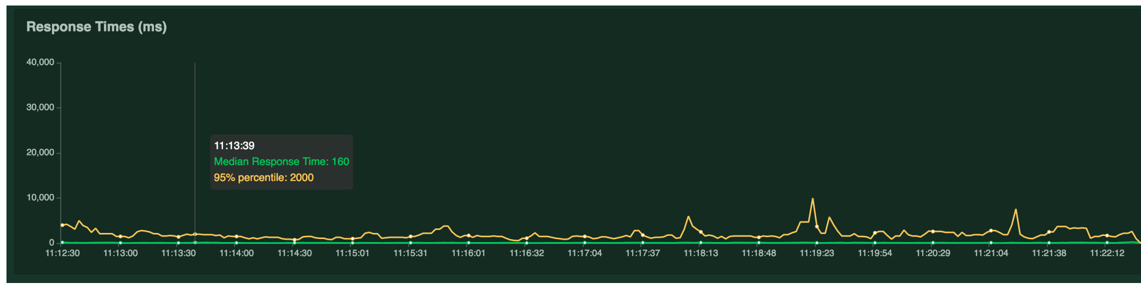

Figure C.8: Locust statistics dashboard for load of size 100 users, without path finding requests and including a node limit of 10.

.

Appendix D

Terminology

- Bitcoin : (Upper-case B) Referring to the protocol rather than the currency
- bitcoin : (Lower-case b) Referring to the units of the cryptocurrency
- Entity: An entity refers to companies and services operating on bitcoin, such as exchanges, pools, gambling sites etc.
- Clustering: When using the term *clustering*, we refer to the process of grouping together Bitcoin addresses that are assumed to belong to the same user, which may be an individual bitcoin user or an entity.
- Explorer tools: To assist with referencing particular software components of this project, each core software component has been given a name, which is quite arbitrary but following the theme of names of tools explorers used to use to navigate.