universität
innsbruck

Master Thesis

Generalizing BlockSci to Cross-Chain Analyses of Forked Ledgers

submitted to the Faculty of Mathematics, Computer Science and Physics of the University of Innsbruck

in partial fulfillment of the requirements for the degree of Master of Science

Martin Plattner, BSc. BSc. (1167741)
martin@mplattner.at

Innsbruck, 15 June 2020

Supervisors: Univ.-Prof. Dr. Rainer Böhme
Michael Fröwis, MSc.

Abstract

Public blockchains such as Bitcoin provide large amounts of financial transaction data. Analyzing this data is of significant interest for scientific research and commercial applications, but also for law enforcement and regulation. Cross-chain analyses enhance established single-chain analyses by including data of multiple chains. Forked chains are particularly suited for cross-chain analyses due to their shared history, which causes direct links between identities across chains. This allows to extract novel insights, such as how user behavior differs across forked chains and how it affects privacy. We identified two main challenges when performing cross-chain analyses of forked chains. First, it is difficult for users to combine the data of multiple chains. We are not aware of a publicly available tool that provides means to efficiently utilize the data relationships between forked ledgers. Second, every additional chain significantly increases the memory footprint and thereby the cost of the analysis. We improve the high-performance blockchain analysis platform BlockSci to overcome these challenges. We add a new multi-chain mode to BlockSci that allows efficient and user-friendly cross-chain analyses of forked chains. This mode has two main features. First, it deduplicates addresses across chains, i.e., addresses are compatible between chains. This is a core requirement of cross-chain analyses and allows to study the activity of the same addresses on multiple chains. Second, the new mode shares common data across chains in memory. This significantly reduces the memory requirement to analyze forked chains. In an extensive evaluation we successfully verify the correctness, performance, and backwards compatibility of the created extension. We use the capabilities of the new multi-chain mode to implement a novel address clustering technique that uses data of multiple chains: cross-chain address clustering. We enhance a Bitcoin clustering with data of Bitcoin Cash and identify over 570,000 additional cluster merges as of Dec 2019. The analysis indicates that certain user behavior, e.g., cash-outs, can compromise privacy across chains. Overall, our extension[1] contributes a robust and extensible foundation for cross-chain analyses to BlockSci.

[1]The code is available on GitHub:
https://github.com/mplattner/BlockSci/tree/a1784376ccb308015c2819dc081a271703b8c5fc

Acknowledgments

Many people supported me during the creation of this thesis. A huge thank you goes to my supervisors Rainer Böhme and Michael Fröwis. Rainer gave me the opportunity to work in a very motivated and highly skilled team. I learned a lot during this time and I am very grateful for this valuable experience. Michael always had good advice that got me back on track when I was stuck. He also spend a large amount of time giving me feedback on the write-up. I also want to thank the rest of the "Information Security and Privacy Lab" and "Dungeon" crew at the University of Innsbruck. They made the time at the office both rewarding and fun. Jakob always made time to help me with his excellent programming and debugging skills. Patrik and Alex also regularly provided helpful feedback.

A big thank you also goes to the BlockSci team at Princeton University, especially Malte Möser and Arvind Narayanan. As a BlockSci developer, Malte gave me very valuable feedback and supported me during multiple seemingly endless debugging sessions. He also did the heavy lifting for our collaboration on the BlockSci paper that we submitted to the 29$^{\text{th}}$ USENIX Security Symposium 2020. Arvind gives me the opportunity to continue my work on BlockSci in the future.

I also want to thank my friends for many helpful discussions, especially Markus Amtmann. Our regular conversations over a "Mango Pago" motivated me when work seemed overwhelming. A big thank you also goes to my family. They have emotionally and financially supported me not only during my work on this thesis, but also all-along my studies in Vienna, Stockholm, and Innsbruck. Finally, many thanks go to my lovely girlfriend Lisa for her incredible support. She always encouraged me and had a lot of patience while I was busy working on this thesis.

This work was supported by two publicly funded projects:

1. *TITANIUM: Tools for the Investigation of Transactions in Underground Markets.*[1]
   (European Union Horizon 2020 grant agreement no. 740558)

2. *VIRTCRIME: Forensic Methods and Solutions for the Analysis of Criminal Transactions in Post-Bitcoin Cryptocurrencies.*[2]
   (Austrian Research Promotion Agency FFG)

[1] TITANIUM project website: https://www.titanium-project.eu/
[2] VIRTCRIME project website: https://virtcrime-project.info/

Contents

1 Introduction

1.1 Motivation

Distributed ledger protocols and the underlying blockchain technology have gained popularity within the last years. This made the analysis of blockchains relevant for many fields, e.g., computer science, economics, and law enforcement. Analyses are facilitated by the fact that the data of open blockchains is publicly available. For example, the popular Bitcoin blockchain provides over 300 GB of financial transactions. Previous research often covered the analysis of only one individual blockchain at a time. However, such single-chain analyses can be extended to a cross-chain analysis by including data of multiple chains. Cross-chain analyses combine the data of multiple chains to enhance the dataset and extract novel insights. From an analysis perspective, forked chains are particularly suited for cross-chain analyses due to their shared history.

A fork describes the creation of a new blockchain, commonly caused by a backwards incompatible change to the protocol of chain A. This results in an additional chain B that branches off chain A at the block height of the fork. The newly created forked chain builds upon the entire transaction history of the parent chain. A popular example of a fork is Bitcoin Cash (BCH), which forked from Bitcoin (BTC) at block height 478,558, i.e., the first 478,558 blocks of BTC and BCH are identical. As a consequence, all addresses that held coins on the parent chain before the fork inherently hold the same amount of coins on the forked chain – locked by the same keys and thus, on the same addresses. The result is a direct link between the digital identities of users across the parent chain and its forked chain. These links can be utilized in cross-chain analyses by including the additional data that forks provide. Such analyses allow to study many open questions, such as how user behavior differs among parent and forked chain, and how it affects privacy. [1]

Blockchain analyses – whether single- or cross-chain – often require to process large parts of the blockchain. This poses several challenges to the analysis platform. From a software perspective, tools that can efficiently handle the graph-structured data of blockchains are needed. Graph databases, like Neo4j, are frequently used for blockchain data analysis [2, 3]. However, such general-purpose databases often do not provide sufficient performance. Using distributed databases is not a viable option as graph data is hard to partition effectively. Instead, specialized high-performance tools are needed. From a hardware perspective, large amounts of memory are needed for blockchain analysis. Keeping most of the relevant data in memory is crucial to achieve reasonable performance. Most cloud providers offer memory-optimized instances with hundreds of gigabytes of RAM, some large instances even come with several terabytes. However, memory is expensive and should be used efficiently to reduce costs. This poses a challenge for

our goal to perform cross-chain analyses across forks. After all, every additional fork significantly increases the dataset and thus the required amount of memory.

These challenges indicate that specialized tools are needed to efficiently analyze blockchain data. One such domain-specific blockchain analysis platform is the popular tool BlockSci by Kalodner et al. [4]. BlockSci addresses and improves upon several weak points of existing analysis tools. It is very fast due to its highly optimized architecture. For example, it can iterate all 500 million transactions of the Bitcoin blockchain in a few seconds. This level of performance is achieved by transforming the raw blockchain data into a custom data layout that is optimized for analysis. A custom in-memory database is used to load the optimized data layout into memory. BlockSci provides a user-friendly Python interface and comes with several useful analysis modules, e.g., to cluster addresses by real-world entities. However, BlockSci's support for cross-chain analyses of forked ledgers is limited. It can currently only load and process the parent and its forked chain individually. Addresses are not deduplicated across chains, which is a core requirement to utilize links across chains in a cross-chain analysis. Although the user could combine and link the data of both[1] chains manually in the analysis script, this is a complex and cumbersome task. The user would experience insufficient performance due to missing cross-chain indices. Loading forked chains also results in a large memory overhead as the shared history of both chains is kept in memory twice. Overall, BlockSci only provides very limited support for cross-chain analyses of forked chains.

1.2 Research Approach

Our long-term goal is to improve BlockSci's support for cross-chain analyses. In this work we aim to create a robust foundation for cross-chain analyses that can be built upon in the future. We limit the scope to analyses across forked chains, as they provide very valuable data due to the links between chains.

We add a new multi-chain mode to BlockSci that allows to load and analyze a chain together with its fork(s). Our focus is on providing high performance, basic means to utilize the links between chains, and a user-friendly interface. A major contribution is the extension of BlockSci's address deduplication to multiple chains. It allows us to provide users with address representations that are compatible across chains, i.e., Address 1 corresponds to the same address on all chains. This unique feature enables many novel cross-chain analyses, such as cross-chain address clustering. We change BlockSci's data layout so that it efficiently supports forked chains. The new layout facilitates sharing data that is common to chains in memory, i.e., the identical blocks of forked chains. This reduces the aforementioned memory overhead when analyzing forks. The user interface is backwards compatible. It provides users with the familiar single-chain experience known from the existing BlockSci version. In this work we do *not* extend the API with cross-chain data retrieval methods. However, the new multi-chain mode is designed with common cross-chain queries in mind and we plan to add a cross-chain API in the future. We evaluate the resulting extension for correctness, performance, and backwards

[1] When we use "both chains" in this work, we refer to a parent chain and its fork, e.g., BTC and BCH.

compatibility. Correctness is evaluated by comparing the returned data of the extended and non-extended versions. Performance is evaluated by benchmarking eight common queries with both versions.

We use the new multi-chain mode to implement a novel clustering technique: cross-chain address clustering. Address clustering uses heuristics to cluster addresses that are likely controlled by the same real-world entity [5]. Address clustering is crucial for law enforcement and blockchain forensics, but is also useful for many other types of analyses. Cross-chain address clustering enhances the established single-chain technique by including data from multiple chains. This allows to create an improved clustering, i.e., with more linked addresses than by using single-chain data alone. Forked chains provide a powerful clustering data source due to the links between identities across chains. Users may be privacy-conscious on the parent chain but perform privacy-harming actions on the forked chain, or vice versa. We use the enhanced clustering module to analyze a cross-chain clustering of Bitcoin and Bitcoin Cash.

1.3 Outline of Contents

Section 2 covers relevant preliminaries: the fundamentals of distributed ledger protocols, the basics of blockchain data analysis, and the relevant parts of BlockSci. Section 3 is the main part of this work and covers the implementation of the new multi-chain mode. We first gather requirements and then describe all required changes in detail. We end with an evaluation of the extension and a discussion of limitations and future work. Section 4 uses the new multi-chain mode to implement a novel address clustering technique that works across chains. We analyze a cross-chain clustering of Bitcoin and Bitcoin Cash. Section 5 concludes this work with a discussion and an outlook.

Contribution Statement

The developers of BlockSci decided to incorporate the new multi-chain mode into a future release of BlockSci. They also invited me, Martin Plattner, to contribute my work to their paper "*BlockSci: Design and applications of a blockchain analysis platform*" [6]. The paper was joint work with Malte Möser[2], Harry Kalodner[2], Kevin Lee[2], Arvind Narayanan[2], Steven Goldfeder[3], and Alishah Chator[4]. I, Martin Plattner, implemented the new multi-chain mode and conducted the cross-chain analysis for Section 3.2 ("*Cashing out on forks hurts privacy*") of the paper. The write-up regarding the multi-chain mode, especially Section 3.2, was joint work. The paper has been submitted to the 29$^{\text{th}}$ USENIX Security Symposium 2020. At the time of writing, the status of acceptance is "minor revision with shepherd".

[2]Princeton University
[3]Cornell Tech
[4]Johns Hopkins University

2 Background

In this section we cover preliminaries that are required to understand the rest of this work. We start with an introduction to distributed ledger protocols in Section 2.1. In Section 2.2 we cover blockchain data analysis, related work, and existing (but mostly insufficient) alternatives to BlockSci. We end with an in-depth coverage of BlockSci's current architecture in Section 2.3. We assume that the reader has a background in computer science. Readers that are experienced with distributed ledger protocols may safely skip large parts of Sections 2.1 and 2.2.

2.1 Distributed Ledger Protocols

In this section we systematically cover the preliminaries of DLPs that are relevant for this work. We follow a bottom-up approach and start with the fundamentals of DLPs in Section 2.1.1. We outline the key challenges and required changes to transition from *traditional ledgers* to decentralized *distributed ledgers*. Many distributed ledger protocols store their data in a *blockchain*, which we cover next. Note that while we approach DLPs is a generic way, we aim to eventually converge to Bitcoin, as it is the main DLP that BlockSci supports. Thus, bear in mind that some details given in Section 2.1.1 are only valid for Bitcoin and similar DLPs. Blockchains commonly contain financial transactions according to a format that is defined in a *transaction model*, which we cover in Section 2.1.2. We focus on the *Unspent Transaction Output (UTXO) model* of Bitcoin. In Section 2.1.3 we cover *forks*, which denote blockchains that split into multiple branches. We outline different types of forks, discuss their persistency, and highlight relevant implications. We end with an in-depth coverage of *Bitcoin*, its data format, its scripting language, common address types, and relevant Bitcoin forks in Section 2.1.4.

2.1.1 Fundamentals

Traditional Ledgers

A traditional ledger is a bookkeeping tool to record a chronological sequence of events in a consistent and transparent manner. Ledgers are often used to record financial transactions[1], e.g., bank transfers. A typical transaction could be: "Transaction #5 (on 2019-05-23 at 10:40pm): Alice sends Bob USD 4.99". Ledgers require integrity and a high level of transparency to allow traceability and auditability. The following two characteristics support this requirement by limiting the means of making changes to the ledger. First, a ledger is usually managed by one central authority, e.g., the bank. This

[1] Throughout this work we assume ledgers that are used to store financial transactions.

ensures that only one clearly defined party is in control and thus, responsible. The central authority can deploy access control systems so that only authenticated and authorized entities can access the ledger. Second, ledgers are often append-only, i.e., new entries can be added but existing entries are never modified. While having a central authority simplifies the administration of the ledger, it also requires that all participants trust the central authority. For example, all customers of a bank must trust the bank that it handles the ledger with due diligence to avoid incidents like a loss of funds. This trust requirement is not desired in some use cases. For example, multiple competing companies may want to maintain a ledger for their business sector, but can not agree on the central authority because of trust issues. Another example are people who tend to distrust centralized institutions, e.g., national banks, and prefer a decentralized system instead. These and other motifs led to a large research effort to make centralized ledgers decentralized. Bitcoin was the first system that solved this issue for open systems, i.e., systems where anyone can join and the authorities are not predefined. Since its release in 2009 such decentralized payment systems are commonly referred to as Distributed Ledger Protocols (DLPs).

Distributed Ledgers

DLPs improve traditional ledgers by removing the requirement to trust a central authority. There is no community-wide accepted definition of DLPs. We describe DLPs in the sense of Bitcoin in the following. Such DLPs do not require a central authority. Instead, an arbitrary number of untrusted parties can "jointly" manage and validate the distributed ledger. The participants of the DLP – also called nodes – create a peer-to-peer network to communicate with each other. The network is open and permission-less: nodes can join, leave, and rejoin the network at any time without asking for permission. No strong identities are tied to individual nodes. Thus, every party can create an arbitrary number of identities. This open organization implies that nodes do not necessarily trust each other – they might not even know each other. However, it is still required that all nodes agree on a valid and consistent state of the ledger. This includes that all transactions of the ledger are in the correct order and valid. Transactions are invalid if they try to transfer coins that the sender does not have, or if they try to spend already spent coins, known as double-spending. This high level of openness is an essential property of DLPs. All solutions prior to Bitcoin assumed a less open system than described above. DLPs try to solve this issue: reaching (eventual) consensus on an open network with weak-identity nodes that do *not* trust each other.

Consensus in an Open Network

Reaching consensus in an open network requires a mechanism that ensures that all nodes agree on a consistent and valid state – a so-called distributed consensus protocol. We first outline a rather abstract model of distributed consensus protocols: imagine a paged ledger, e.g., a physical notebook that is used to record transactions chronologically[2].

[2]Later on, we replace the notebook with a blockchain where every block represents a page.

While the central authority is clearly defined for traditional ledgers, it is not so trivial for distributed ledgers. Who should be the authority that is allowed to add new transactions to the ledger? Who decides which transactions are valid and therewith contributes to the current state of the ledger? All participating nodes represent possible authority candidates and all want to occupy this powerful role. DLPs use a distributed consensus protocol to solve the issue of selecting a leader among all participants. In practice, many protocols aim to select a new leader "randomly", an approach known as random leader selection. This allows that power is distributed among all validating nodes, which are also called miners. A widely used random leader selection mechanism is proof of work (PoW). Another common mechanism is proof of stake (PoS). However, we only cover PoW in the following[3] as it is sufficient to understand the rest of this work. PoW requires miners to solve a computationally hard puzzle in order to propose a new page for the ledger. A page contains a set of transactions that the miner can select freely out of the transactions that were recently broadcasted on the network. As a consequence, adding a page to the ledger becomes expensive as it is tied to a real-world resource: electricity, mining hardware etc. Most current implementations of PoW require the miner to find a cryptographic hash value that is below a certain threshold. Miners have no other option than to guess (=brute force) the correct hash. They do so by repeatedly hashing the page to be validated with an appended nonce that is changed for every attempt. One might wonder why miners participate in the validation process. After all, they have to pay for the mining hardware and the electricity to operate it. DLPs have an incentive model that tries to ensure that miners validate and behave according to the protocol. As soon as a miner finds a solution – the nonce for the proposed page – they immediately broadcast it to the network. While the miner needed a lot of computational power to find the correct hash, other nodes can quickly verify if the solution is correct. If the other nodes accept the page as valid, they add it to their version of the ledger and the miner gets a reward. The possibility to reward nodes for fair behavior is an essential feature of DLPs. The fact that DLPs have an intrinsic unit of value – the unit of the ledger – facilitates paying rewards to users. DLPs that mainly use the unit to implement a virtual currency are referred to as cryptocurrency. Rewards that are given to miners represent the monetary supply of a cryptocurrency, similar to a central bank that prints money. As soon as the other miners receive the new valid page, they know that they lost the current round of the puzzle. A new round starts immediately: all miners stop looking for a solution to their current individual candidate page, assemble a new page, and start the solving process again. You can think of this repeated process as a miniature lottery where the prize is the reward and the permission to add a page to the ledger. More formally, this process is also called "probabilistic block race", as the likelihood that a miner finds the solutions is proportional to her resources, e.g., the hash rate. The difficulty of the PoW puzzle can be adjusted by changing the threshold that the hash is compared to. Most DLPs do globally adjust the difficulty so that new pages are added in regular intervals on average. For example, the intended block time of Bitcoin is 10 minutes, meaning that a new round of the miniature lottery starts every 10 minutes on average. Besides the

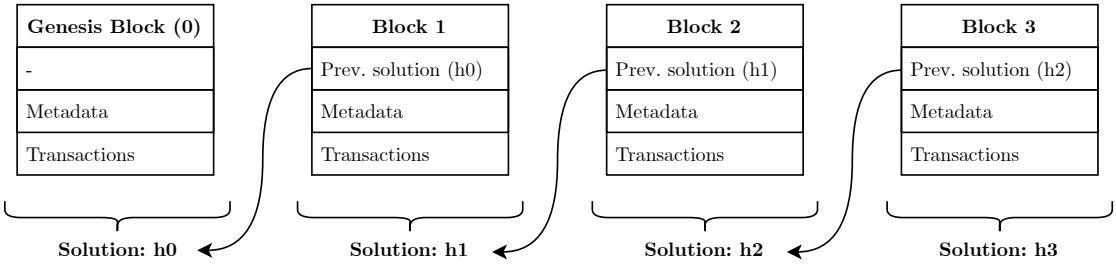
[3]Note that we cover PoW as it is implemented in Bitcoin.

benefits for the security of the DLP, this regular process results in a predictable supply of new coins. The outlined protocol provides a probabilistic selection of nodes that are allowed to propose a new page for the ledger with an integrated self-enforcing rate limit. In order to further discuss DLPs and their security we need to discuss a fundamental data structure first: the blockchain.

Blockchain

Most DLPs store the data of the ledger in a blockchain data structure[4], as shown in Figure 2.1. The terms blockchain and blockchain system are commonly used as synonyms for DLPs. A blockchain is an authenticated data structure that consists of a ordered set of blocks. We can think of a block as a page of the physical notebook-based ledger in the previous section. The number of the block is also called block height, e.g., the 5^{th} block has block height 5. Every block contains a block header with metadata, and a ordered set of transactions. The concrete data structure of blocks and transactions are implementation-specific for individual DLPs. Bitcoin's data structures are outlined in Section 2.1.4. The first block of a blockchain is called genesis block and is hardcoded in the source code of the DLP implementation. Each additional block contains a cryptographic hash of its previous block in the block header. This hash represents the solution that was found by the miner using the PoW mechanism. The result is a cryptographically linked ("chained") list of blocks – hence the name blockchain.

Figure 2.1: A blockchain represents a ordered set of cryptographically linked blocks.



The purpose of this authenticated list is immutability. If an existing block in the blockchain is modified, its hash changes. Thus, the link from the next block to the changed block becomes invalid. The result is that changing an existing block invalidates the links among all blocks that come after it. However, it is only the combination of the blockchain data structure with PoW that provides actual security. Recall that mining a block requires a substantial amount of computational work. In order to rewrite the history of a blockchain an attacker has to mine upon an existing block and provide a valid set of blocks so that his blockchain is higher than the current public version of the chain. It has to be higher because most DLPs resolve such conflicts by always selecting the longest available valid branch of the chain. The longest chain equals to the chain that

[4] An alternative data structure for DLPs are directed acyclic graphs, which are for example used by a DLP called IOTA. However, non-blockchain-based DLPs are not within the scope of this work.

has most work put in it. As all blocks build on each other, the amount of work put into the blockchain accumulates. This makes it increasingly harder and more expensive for an attacker to provide a valid set of alternative blocks and thus, increasingly unlikely to be successful. Therefore it is recommended by most DLPs to wait for several new blocks before considering a recorded transaction as confirmed. Every new block represents an additional confirmation of all previous blocks. For Bitcoin it is common to wait for six confirmations, which on average takes 60 minutes.[5] After enough confirmations the state of a DLP can practically be considered immutable and consistent, with the limitation that both properties are only achieved *eventually* and not immediately.

2.1.2 Transaction Model

A common use-case for traditional ledgers is tracking financial transactions and the balance of accounts over time. This is also true for DLPs, which have to track the transactions and balances of its participants. DLPs commonly use digital signatures based on public-key cryptography to manage identities and ownership. Every node can have one or more key pairs, consisting of a public and a private key. Generating new keys is a cheap operation and a single key pair can also be used to derive an arbitrary number of new key pairs from it. Thus, every user of the system can participate with an arbitrary number of identities. The public key is used to receive coins, and the private key is used to spend previously received coins. Most DLPs transform the public key to a standardized address format before it is used to receive coins. Addresses are more user-friendly than using public keys directly as they include a checksum and exclude ambiguous characters to avoid typing errors. Bitcoin has several address types[6] which are outlined in Section 2.1.4.

The specifics of transactions are defined in a transaction model. Two dominant transaction models are used in DLP systems: the Unspent Transaction Output (UTXO) model, and the account model. The account model is more intuitive as it is similar to the traditional banking system where a user has just one or a few accounts. It is primarily used by DLPs that support smart contracts, e.g., Ethereum (ETH). As BlockSci only supports UTXO-based DLPs, we only cover the UTXO model in the following. The UTXO model is more complex than the account model, but better protects the privacy of the participants. It is used by many DLPs, including Bitcoin and its derivatives, e.g., Bitcoin Cash, Dash, and Litecoin. Transactions transfer value between users. They consist of inputs and outputs. A common analogy is that transaction inputs specify where the coins of the transaction come from, and of the transaction outputs specify where the coins go to. Outputs are locked by a spending condition that is part of the output and defined by the sender. A common spending condition is “pay to whoever can provide a

[5] The number of required confirmations depends on the block time of the DLP and the value at risk, e.g., the amount of the transaction. A transaction for a coffee may require less confirmations than for a Lamborghini.

[6] For example, 1BvBMSEYstWetqTFn5Au4m4GFg7xJaNVN2 is a Pay-to-Public-Key-Hash (P2PKH) address.

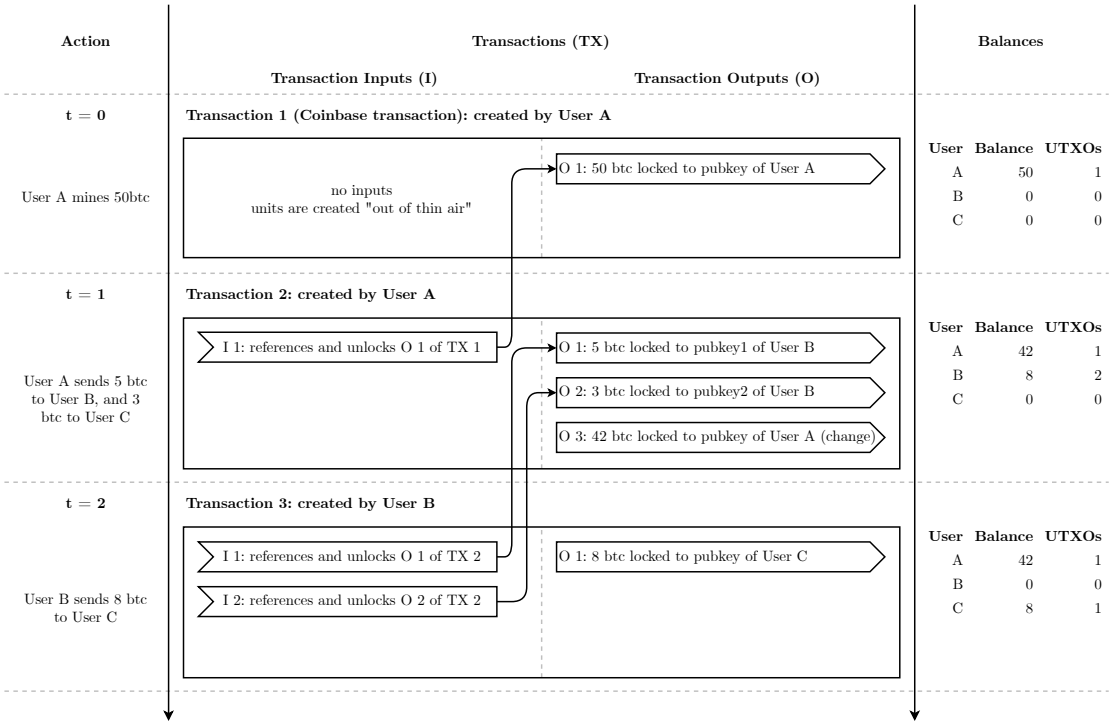
valid signature for the attached public key"[7]. Inputs spend previously created outputs by referencing and unlocking them. Unlocking means to fulfill the spending condition, e.g., providing a valid signature for the public key that is attached in the referenced output. The spending condition and its fulfillment are provided via scripts in the output respectively input. Most DLPs use a custom scripting language for this purpose. For example, Bitcoin uses a language called Script, which we cover in Section 2.1.4. The consensus rules enforce that every output can be referenced only once by an input. This prevents that outputs are spent multiple times, known as double-spending. An output is indivisible, i.e., the entire value of the output has to be spent.

There is a special type of transaction that does not have any inputs – the coinbase transaction. The coinbase transaction is created by the miner of the block. It pays the miner of the block the reward and thus, does not need any inputs. Instead, the units are coming from nowhere, e.g., "out of thin air". In Bitcoin, the coinbase transaction is the first transaction of each block.

Figure 2.2 shows how the UTXO model works using three sample transactions. We assume User A is a miner and Users B and C are regular users. Transaction 1 is a coinbase transaction created by User A where she pays (=locks) the mining reward of 50 btc to herself. The input of Transaction 2 unlocks the output of Transaction 1 and sends (=locks) 2 outputs with a total value of 8 to User B. Note that the entire output of Transaction 1 is spent, and the rest (42 btc) is sent back to User A in a "change output". In Transaction 3, User B unlocks the received outputs and creates one output locked to User C. The right side shows the number of UTXOs and the accumulated balance for each user (after applying the transaction on the left).

[7]This condition is referred to as Pay-to-Public-Key (P2PK).

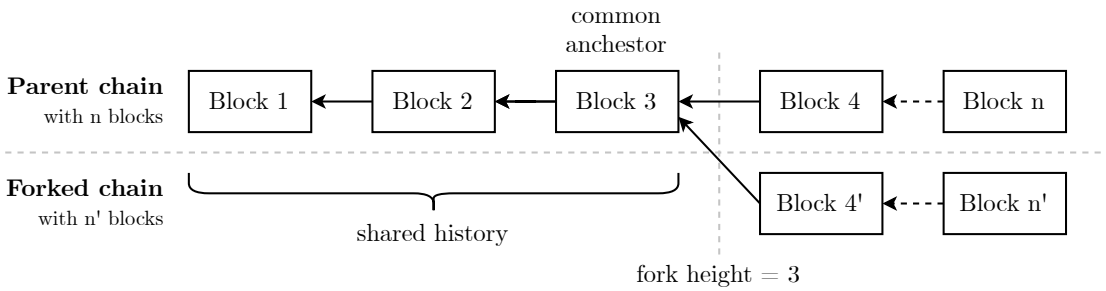
Figure 2.2: A sample flow of coins using the UTXO model.



2.1.3 Forks

Under normal conditions a blockchain grows linearly as miners add block after block to the end of the chain. This linearity is broken when multiple blocks are added to the last block, as shown in Figure 2.3. In this case we speak of a fork ("the blockchain forked"), or a chain split.

Figure 2.3: Illustration of a fork.



The concept of forks is essential for this work. Forks occur if miners propose multiple new valid blocks that are mined upon the same common ancestor block. Depending

on the type of the fork this may be desired, e.g., to update the consensus rules of the DLP; or undesired, e.g., when the fork is caused by inconsistencies due to network delays. In any case the result are multiple branches of the blockchain. Forks share a common history with the parent chain. This is important to understand: if a blockchain splits into two branches, e.g., at block height 3, both branches share the identical history up to block 3, as illustrated in Figure 2.3. The shared history makes forks interesting for cross-chain analyses that combine data of parent and forked chain. We can differentiate between several types of forks. However, there is no community-accepted and widely used terminology regarding forks. A recent working paper addresses this issue and provides a classification framework for forks [7]. In the following we cover two common types of forks.

Natural Forks

The decentralized nature of a DLP can cause inconsistencies in the current state of the ledger among the participating nodes. The state on nodes may vary due to delays or packet loss on the peer-to-peer network. This can cause temporary branches of blocks that we call natural forks. Natural forks are also referred to as "*accidental fork*" [8, p. xxv] and "*unintentional process-based fork*" [7] in literature. A natural fork commonly occurs when in one round two miners find the solution for their candidate block at roughly the same time. The result is that both new blocks build upon the same parent block. This is a problem because both blocks might contain conflicting transactions, e.g., double-spending of the same coins. The consensus protocol has to arbitrate such conflicts to ensure a single and consistent state of the ledger. Most DLPs resolve natural forks automatically, e.g., by implementing a rule that nodes always accept the longest current branch as the valid chain. As soon as a miner proposes another new block mined upon one of the two branches, one branch becomes longer and is accepted as the valid chain. As a consequence, the other branch and its blocks are wiped out.

Protocol Update Forks: Soft and Hard Forks

Another scenario where forks commonly occur is when the consensus protocol of a DLP is updated. This includes changes to the rules that determine if a block or transaction is valid. In [7] such forks are called "*deliberate protocol-based fork*". Two update types are commonly seen in practice: soft forks and hard forks. Soft forks are often the preferred option as they do not require legacy nodes to update in order to avoid a permanent chain split. The nature of the changes to the existing rules dictates whether a soft fork or a hard fork is required.

- **Soft forks** update the rules in a way that makes them *more* restrictive, e.g., changing the maximum block size from 2 MB to 1 MB. While the legacy software of soft forks is forward-compatible[8], i.e., legacy nodes will accept blocks created by

[8]Forward compatibility describes that an old version can process inputs intended for a newer version.

updated miners, the updated software is not backwards-compatible[9]. Formally the new set of valid blocks is a subset of the legacy set of valid blocks [7]. Thus, new blocks mined according to the updated rules are also accepted as valid by legacy nodes. This allows to update a DLP without requiring all nodes to update their software (immediately). Soft forks are regularly used in practice to update the Bitcoin system, e.g., in BIP16[10] that added the Pay-to-Script-Hash (P2SH) address type[11].

- **Hard forks** update the rules in a way that makes them *less* restrictive, e.g., changing the maximum block size from 2 MB to 4 MB. While the updated software of hard forks is backwards-compatible, i.e., updated nodes will accept blocks created by legacy miners, the legacy software is not forward-compatible. Formally the new set of valid blocks is a superset of the legacy set of valid blocks [7]. Thus, new blocks mined according to the updated rules may be seen as invalid by legacy nodes. A hard fork is what people commonly refer to by the term "fork". One prominent example of a hard fork is Bitcoin Cash, which split from the original Bitcoin chain in August 2017 over disagreement about the maximum size of blocks [9].

Persistence of Forks

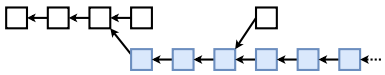
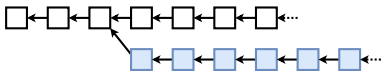
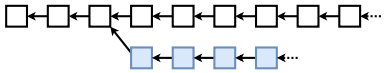
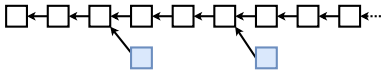
An important concept for this work is the persistence of forks. Forks can be sustained permanently or temporary. Our scope are permanent forks, as they provide data that is permanently persisted in the blockchain. In contrast, in temporary forks one of the competing branches wins and the other branch is removed, i.e., the blocks and transactions are wiped out. While temporary forks could be analyzed as well, their short lifetime and thus small size does not require the extension we are implementing in Section 3.

Natural forks are resolved by the consensus protocol with the rule that all nodes accept the longest current branch as valid. Thus, natural forks are temporary and not within the scope of this work. For soft forks and hard forks it is somewhat more complex. The aim of both types is commonly that the updated rules become the new default, i.e., the new branch should ultimately replace the old branch. This is to avoid a permanent chain split, which can severely damage the reputation and value of the DLP, as it confuses users, causes uncertainty, and may even cause security issues. Avoiding a chain split requires the support of miners and users – miners need to allocate their hash rate, and users need to update their software. However, in practice the changes that forks introduce are often highly controversial, leading to a competition of both the legacy and the updated version. In theory the outcome mainly depends on the allocation of consensus-relevant resources, i.e., the hash rate, as shown in Table 2.1 [7]. We denote the hash rate supporting the old respective new rules with r_{old} and r_{new}. For a soft fork to replace the legacy branch, a majority of the hash rate needs to be allocated to mining according to the new rules, i.e, $r_{new} > r_{old}$. Then the branch supported by updated miners grows quicker and may

[9] Backward compatibility describes that a new version can process inputs intended for an older version.
[10] Bitcoin Improvement Proposal (BIP)
[11] Bitcoin's address types are covered in Section 2.1.4.

wipe out the legacy branch. In this case there is no permanent chain split and the soft fork was successful. On the other hand, if a majority of the hash rate sticks to the legacy rules, i.e., $r_{new} < r_{old}$, a permanent chain split may be the result. For hard forks it is reversed, as shown in Table 2.1.

Table 2.1: Persistency of soft and hard forks based on the allocation of consensus-relevant resources r_{new} and r_{old}, i.e., the hash rate. White blocks are mined according to the old rules, and blue blocks according to the new rules. [7]

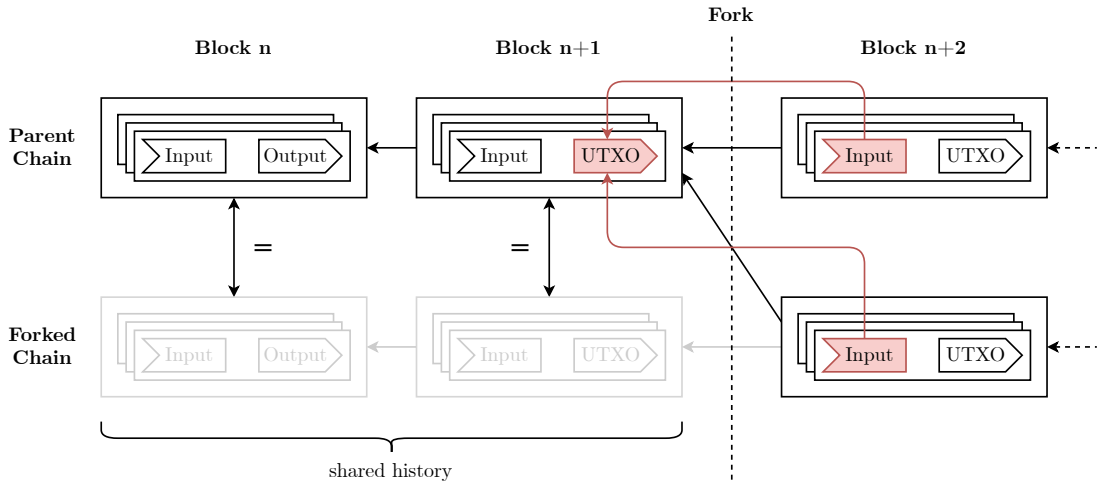
	soft fork	**hard fork**
$r_{new} > r_{old}$	 old branch fails, soft fork wins	 both win, permanent chain split
$r_{new} < r_{old}$	 both win, permanent chain split	 old branch wins, hard fork fails

Note that this is a rather formal classification that assumes all nodes receive and consider both old and new blocks. Only then we can observe the described competition of the old and the new branch. In practice, this is often not the case and thus, the persistence is also dependent on the specific implementation. For example, the Bitcoin Cash specification required that the first forked block must be larger than 1 MB so that the post-fork Bitcoin chain (with blocks smaller than 1 MB) is invalid and thus, a wipe-out of the fork is impossible [10, Req. 3]. That said, almost all hard forks are permanent in practice. In the following we concentrate on permanent forks only.

Implications of Permanent Forks

The fact that forked ledgers share a common history has some very relevant implications. Recall that all pre-fork blocks are identical. First, this implies that both chains share all pre-fork addresses. Second, and more importantly, all outputs that are unspent at the fork height can be spent in *both* chains, as illustrated Figure 2.4. For the users this means that all their coins on the parent chain are "duplicated" on the forked chain. These duplicated coins in the form of UTXO can be spent on both the parent and the forked chain. The UTXOs are inherently locked by the same addresses on both chains. Thus, forked ledgers have direct links between identities across chains. This makes permanent forks a powerful data source for cross-chain analyses.

Figure 2.4: Pre-fork UTXOs can be spent on both the parent and the forked chain.



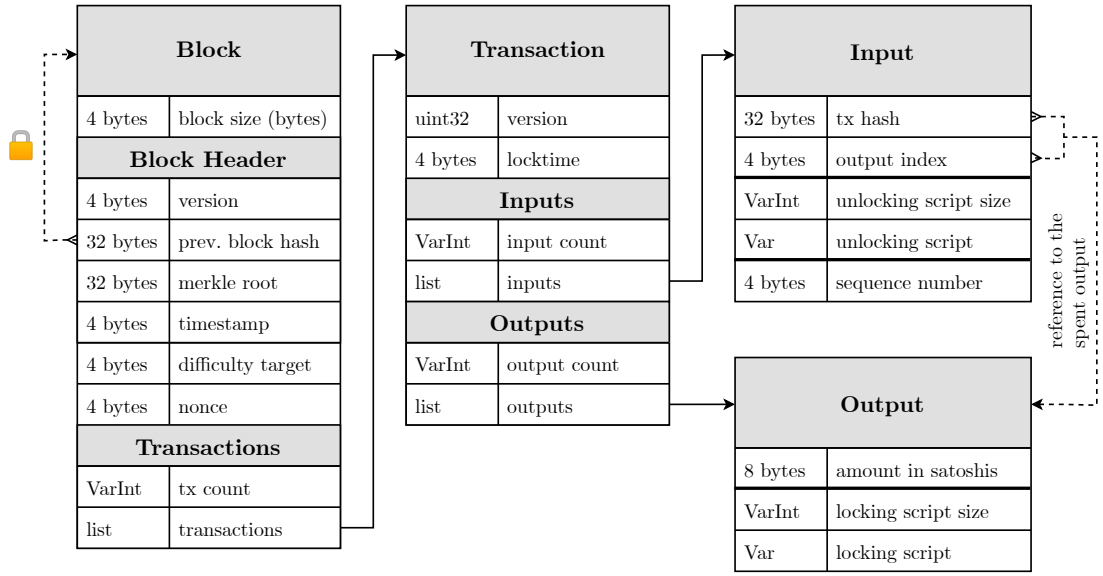
2.1.4 Bitcoin

Bitcoin is known as the first cryptocurrency. We cover Bitcoin in more detail because BlockSci only supports DLPs that use the same data format as Bitcoin. The white paper of Bitcoin was published in 2008 under the pseudonym Satoshi Nakamoto [11]. A couple of months later the Bitcoin blockchain went live in January 2009. Bitcoin became known to the general public due to widespread media coverage following the significant price increases in 2017. With a 67% share of market capitalization among more than 5,400 cryptocurrencies listed on CoinMarketCap.com [12] as of May 2020, Bitcoin can be considered the most valuable cryptocurrency. Most concepts of the previous sections are used by Bitcoin: data is stored in a blockchain, PoW-based mining with a target block time of 10 minutes is used for the random leader selection, and the UTXO model is used as the transaction model.

Data Format

Figure 2.5 shows the data layout of Bitcoin. The top-level data structure are blocks, which contain transactions, and transactions contain the inputs and outputs of the UTXO model. All data structures have several metadata fields. For example, blocks contain a timestamp, the nonce used by the miner, and the previous block hash. This hash cryptographically links each block with the previous block to create the blockchain. An in-depth coverage of all fields can be found in [8]. Inputs reference the output that they spend using the `(tx hash, output index)` tuple. Both inputs and outputs contain a script written in the Bitcoin Script language.

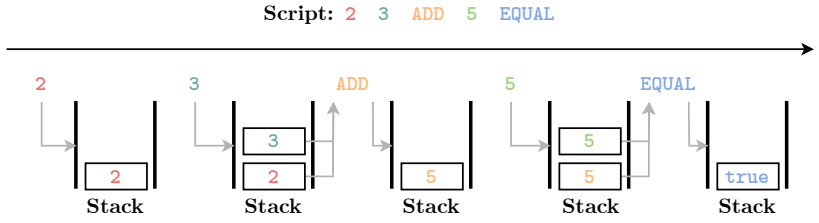
Figure 2.5: Data format of Bitcoin [8]. Dashed lines represent links between blocks and from inputs to the output they spent.



Bitcoin Script

Bitcoin uses a custom domain-specific language called Script to define spending conditions. Every output defines a spending condition in Script, and the input that spends the output must provide a valid script to fulfill this condition. Script is a simple stack-based language that is deliberately not Turing-complete. This increases predictability and security, e.g., prevents denial-of-service attacks. It supports several instructions called opcodes[12]. Script uses reverse-polish notation, i.e., the operands come first and then the operator. Figure 2.6 illustrates the execution of the script `2 3 ADD 5 EQUAL`. It adds two numbers (2 and 3) and checks if the result is equal to the expected sum (5). The numbers represent data and are pushed onto the stack. The opcodes `ADD` and `EQUAL` pop operands from the stack and push back the result (`true`). [8]

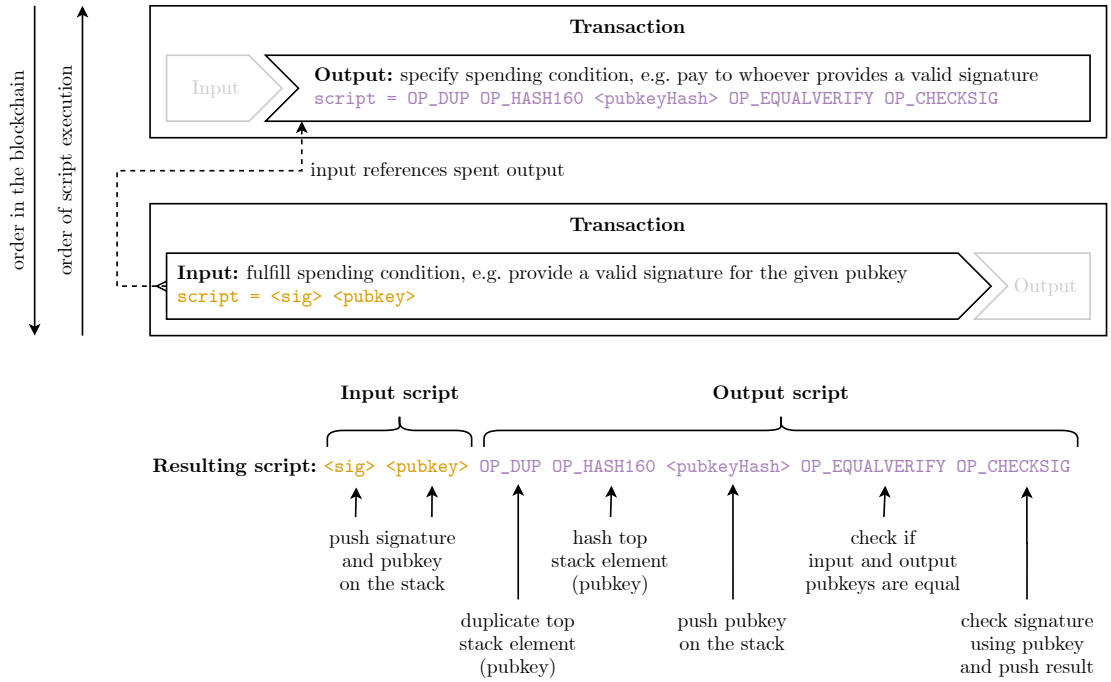
Figure 2.6: Step-by-step execution of a simple script. [8, p. 135]



[12] A list of all supported opcodes can be found at [13].

The example above illustrates the mechanics of Script. However, more involved scripts are needed to create transactions that transfer coins between users. Bitcoin transactions are validated by evaluating the scripts for every input-output pair of the transaction. The scripts of the spent output and the spending input are combined as shown in Figure 2.7. The input script comes first, followed by the output script. The combined script is evaluated and the result determines whether the transaction is valid or not. Transactions are valid if the top element on the stack is true or any other non-zero value, or if the stack is empty after the execution [8, p. 134].

Figure 2.7: Bitcoin combines the input and output scripts to validate transactions.



Address Types

Bitcoin supports several pre-defined script patterns, also known as address types. The most common types are Pay-to-Public-Key-Hash (P2PKH)[13], Pay-to-Script-Hash (P2SH), and Multisig. All of them transfer coins from one address to another. The `OP_RETURN` or *nulldata* type creates outputs that store arbitrary data in the blockchain. Such outputs can not be spent and the contained coins are irreversibly lost. All other scripts that do not follow any of these patterns are called non-standard scripts.

When Bitcoin was released the Pay-to-Public-Key (P2PK) address type as shown in Listing 2.1 was frequently used. The sender provides the full public key of the recipient in the output. The spending input must provide a valid signature for the output's public

[13] And its simpler but less frequently used Pay-to-Public-Key (P2PK) variant.

key, i.e., the spending condition is "pay to whoever can provide a valid signature for the attached public key". When evaluating a P2PK script, the signature of the input is pushed onto the stack first, followed by the public key of the output. The `OP_CHECKSIG` opcode pops both values of the stack and checks whether the signature is valid.

Listing 2.1: Pay-to-Public-Key (P2PK) script

```
input  = <sig>
output = <pubKey> OP_CHECKSIG
```

The Pay-to-Public-Key-Hash (P2PKH) script shown in Listing 2.2 is similar to the P2PK type and largely superseded it. Instead of the public key it provides a hash of the public key in the output, i.e., the spending condition is "pay to whoever can provide a valid signature for the attached hashed public key". During script evaluation, the input's `<pubKey>` is duplicated with `OP_DUP` and then hashed using `OP_HASH160`. This hashed version of the input's `<pubKey>` is compared to the `<pubKeyHash>` of the output using `OP_EQUALVERIFY`. If they are equal, `OP_CHECKSIG` is used to verify the signature.

Listing 2.2: Pay-to-Public-Key-Hash (P2PKH) script

```
input  = <sig> <pubKey>
output = OP_DUP OP_HASH160 <pubKeyHash> OP_EQUALVERIFY OP_CHECKSIG
```

Another common address type is Pay-to-Script-Hash (P2SH), as shown in Listing 2.3. P2SH outputs provide the hash of the script that can unlock the output, i.e., the spending condition is "pay to whoever provides a script that a) matches this hash and b) executes successfully". While P2SH addresses do not require to use signatures, it is common and recommended to do so. P2SH input scripts can wrap another address by setting the `redeemscript` to the script pattern of another type, e.g., `redeemscript = <sig> <pubkey> OP_CHECKSIG` to wrap a P2PK address. P2SH addresses are often used to wrap a multi-signature address.

Listing 2.3: Pay-to-Script-Hash (P2SH) script

```
input  = <sig> <redeemscript>
output = OP_HASH160 <HASH160(redeemscript)> OP_EQUALVERIFY
```

Multi-signature addresses, commonly called multisig addresses, provide N public keys in the output and M valid signatures must be provided in the input that spends the output. This is known as a M-of-N scheme, as in "pay to whoever provides M signatures for the given set of N public keys". This facilitates that coins are controlled by multiple parties. For example, an enterprise could protect their coins so that at least two of (chief executive officer, chair, head attorney) have to sign transactions.

Listing 2.4: Multi-signature script

```
input  = OP_0 <signature A> [<signature B>] [< signature M>]
output = <M> <pubkey A> [<pubkey B>] [<pubkey N>] <N> OP_CHECKMULTISIG
```

Bitcoin Forks

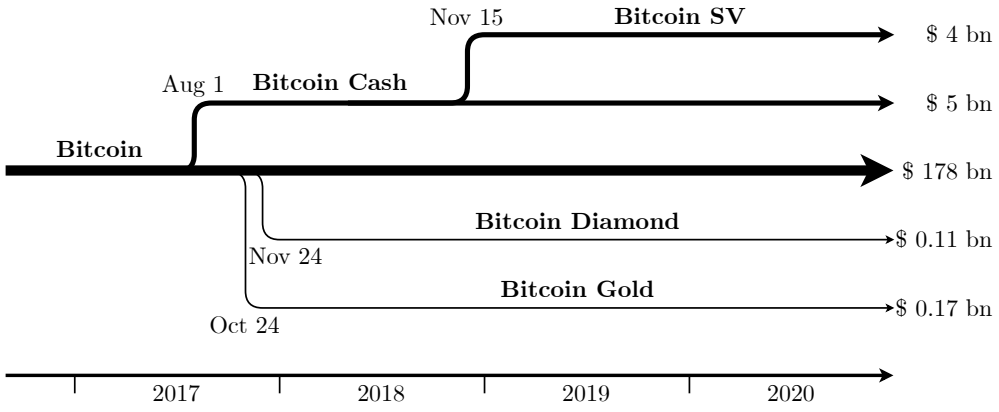
The popularity of Bitcoin has caused the emergence of more than 50 forks [14]. Some of these forks emerged due to disagreement over the future of Bitcoin. A prominent example is Bitcoin Cash, which forked off Bitcoin in August 2017 at block height 478,558, after controversy regarding the maximum block size. This debate continued and led to the Bitcoin SV fork that branched off Bitcoin Cash in November 2018. Most of Bitcoin's forks can be considered failed and are irrelevant. This is backed by the fact that CoinMarketCap [12] only lists 10 of the 50 forks listed on [14], see Table 2.2. We conjecture many of the failed forks were mainly created to take advantage of Bitcoin's name and prominence. Figure 2.8 shows a timeline with all forks that have a market capitalization over USD 100 million.

Table 2.2: Bitcoin forks listed on CoinMarketCap [12]

Rank[1]	**Coin**		**Fork**		**Market cap.**
	Name	**Symbol**	**Height**	**Date**	
5	Bitcoin Cash	BCH	478,558	2017-08-01	$ 4,935,841,838
6	Bitcoin SV[2]	BSV	556,766	2018-11-15	$ 3,903,660,757
39	Bitcoin Gold	BTG	491,407	2017-10-24	$ 173,592,624
57	Bitcoin Diamond	BCD	495,886	2017-11-24	$ 109,539,201
294	MicroBitcoin	MBC	525,000	2018-05-28	$ 11,871,814
814	Bitcoin Atom	BCA	505,888	2018-01-24	$ 1,362,332
1486	Bitcoin Interest	BCI	505,083	2018-01-20	$ 89,215
2176	Super Bitcoin	SBTC	498,888	2017-12-12	–[3]
2339	BitcoinX	BCX	498,888	2017-12-12	–[3]
2442	Bitcoin God	GOD	501,225	2017-12-27	–[3]

[1]: by market capitalization on CoinMarketCap [12] as of May 9, 2020
[2]: forked from Bitcoin Cash, –[3]: market cap. not available

Figure 2.8: Bitcoin forks with a market cap. over $ 100 million as of May 9, 2020 [12].



2.2 Blockchain Data Analysis

Public blockchains provide a very large dataset of financial transactions. The Bitcoin blockchain alone is almost 300 GB as of May 2020. Blockchain data analysis can help to better understand these new systems, detect trends, but also identify potentials and risks. The scope of this work are analyses that not only analyze external blockchain data, e.g., prices, but (also) include data from the ledger itself. In the following we summarize a selection of prior blockchain data analysis research. Many of the available works cover privacy, cybercrime, or user behavior.

The public nature of blockchains can pose a privacy risk to the users. This has already been shown in 2013 for the Bitcoin blockchain [5]. Albeit users can generate an arbitrary number of addresses, they can be clustered by users using heuristics. We cover address clustering in more detail in Section 2.2.1. Blockchains like Monero and ZCash are known for their privacy-enhancing features. Möser et al. empirically analyze traceability in the monero blockchain [15]. They find that for 62% of all transaction inputs with mixins the real input can be deduced by elimination in a "chain-reaction" analysis. Kappos et al. evaluate the privacy of ZCash [16]. They find that most users do not take advantage of ZCash's privacy features, and the users who do are using the features in a wrong way such that the anonymity set can be decreased significantly.

Due to a higher level of privacy, DLPs are frequently used for criminal activities. This includes payments on darknet markets, ransomware payments, and money laundering. Vasek et al. [17] empirically analyze cryptocurrency scams such as mining scams, scam wallets, fraudulent exchanges, and Ponzi schemes. They found that in all 192 analyzed scams at least USD 11 million have been paid by 13,000 distinct victims. Huang et al. [18] did a similar analysis for ransomware payments. They found that for the analyzed ransomwares, 19,750 potential victims paid over USD 16 million ransom over a two-year period.

Bartoletti et al. [19] study the usage of Bitcoin's `OP_RETURN` opcode that can store arbitrary data in the Bitcoin blockchain. They detected several protocols that serve as a digital notary or provide ownership certificates for an asset.

2.2.1 Address Clustering

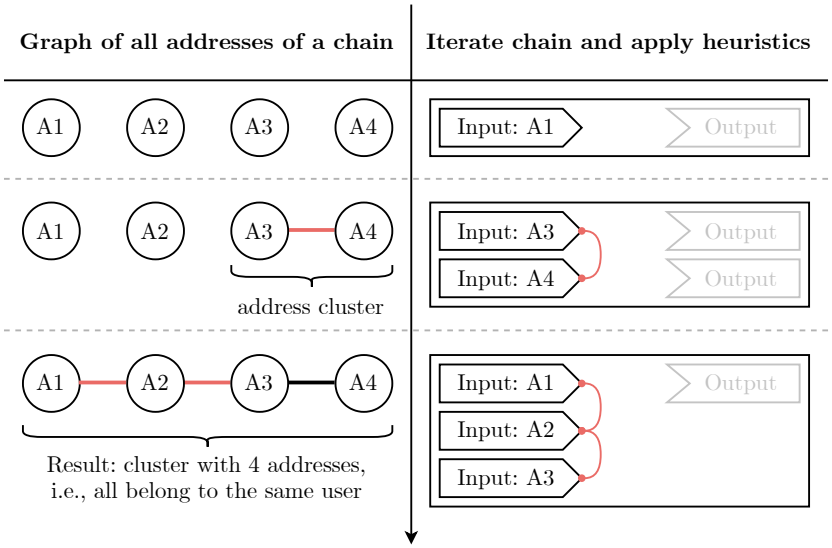
Blockchain transactions transfer coins from one address to another. While every user could use a single address for all activities, this is not the case in practice. Instead, most wallets create a new address for every transaction to protect the privacy of the user. This complicates law enforcement investigations but also all studies that aim to analyze data on a per-user level. Address clustering can mitigate this issue to some extend. It uses heuristics to link addresses that are likely controlled by the same user or entity. Two heuristics are commonly used: multi-input and change address detection. [5, 20]

1. The **multi-input heuristic** assumes that all inputs spent in the same transaction are controlled by the same entity. The underlying assumption is that a transaction is usually created by a single user. Thus, for a given transaction the heuristic links the addresses of all inputs. [5]

2. The **change heuristic** tries to identify the change address of a transaction, e.g., based on client software or user behavior. The underlying assumption is that the change output is locked to an address that is controlled by the creator of the transaction. From the multi-input heuristic we known that the creator likely controls all input addresses. Thus, the change address can be linked with the input addresses. [5, 20]

The heuristics can be applied on all transactions of a blockchain to create a graph of addresses. In the resulting clustering, all connected addresses should belong to the same user. An address clustering is inherently imperfect, as the heuristics may produce false positives, i.e, spurious edges, and false negatives, i.e., missing edges. Figure 2.9 shows how the multi-input heuristic is applied over multiple transactions. BlockSci provides a clustering module to automate this process, see Section 2.3.4. In Section 4 we present an enhanced address clustering technique that works across chains.

Figure 2.9: Using the multi-input heuristic to link addresses.



2.2.2 Cross-Chain Analyses

The term cross-chain or cross-blockchain was previously used primarily in the context of cross-chain interoperability, e.g., to enable atomic cross-chain transactions [21, 22]. Cross-chain data analyses combine data of multiple chains. This can improve existing single-chain analyses but also allows novel analyses. As cross-chain data analysis is a rather new approach, there is only few prior work and no widely accepted terminology for possible types of cross-chain analyses.

Harrigan et al. [23] examined how the airdrop of the Clams coin to existing Bitcoin, Litecoin, and Dogecoin addresses affected the privacy of the owners. An airdrop is a mechanism to distribute coins or tokens by sending them to random existing addresses.

Airdrops are commonly used to promote a new token by bringing it to the recipient's attention. Harrigan et al. successfully applied cross-chain address clustering using data from the Clams chain to enhance clusters in the Bitcoin, Litecoin, and Dogecoin chains. They conclude that sharing addresses between blockchains poses a risk to the privacy of the user.

Hinteregger et al. [24] used a cross-chain analysis to assess the traceability of Monero transactions. They used data from the Monero blockchain and its hard forks MoneroV and Monero Original to distinguish Monero's decoy inputs from real inputs. They found that Monero's recent protocol updates, e.g., increasing the minimum ringsize, render existing heuristics not more effective than random guessing. They note that this may change in the future as new Monero hard forks might occur that see more activity and thus, provide more data to improve the results.

2.2.3 Existing Tools

Blockchain data analysis has several characteristics that differ from other data analysis tasks. A blockchain can contain hundreds of gigabytes of data and this amount is growing continuously. Such large amounts of data are often stored and processed in a distributed database for performance reasons. But as blockchain data is graph-structured, it is difficult to partition the data effectively. General-purpose databases do not provide sufficient performance [4]. Instead, specialized high-performance tools like BlockSci are needed. Due to the scope of this work we cover BlockSci in greater detail in the separate Section 2.3. In the following we briefly present existing alternatives to BlockSci. However, most of them provide worse performance and/or functionality. While the presented open-source solutions are general-purpose analysis tools or frameworks, most commercial tools have a dedicated use case like forensics, or transaction monitoring to support businesses comply with regulations.

Open-Source Tools

- **BTCSpark** [25] is a Bitcoin analysis platform based on the distributed Apache Spark database. The project website states it "*is currently unmaintained. BlockSci is a similar project with better performance*". According to [4], BTCSpark is up to 8 times slower than BlockSci while needing up to 15 times more resources.

- **BlockParser** [26] is a C++-based blockchain parser for Bitcoin and similar cryptocurrencies. It processes the blockchain linearly and single-threaded. The user can extract data by hooking to several events, e.g., parsing block, or parsing transaction. BlockParser is very slow because it iterates the entire blockchain from disk for every analysis. The project has not been maintained since late 2015. BlockParser is used for the analysis in [17]. According to [4], BlockParser is up to 130 times slower than BlockSci.

- **BitIodine** [27] is a modular Bitcoin parser that builds upon BlockParser. It is written in Rust and has an integrated address clustering feature. As a storage

engine it uses SQLite. The project is unmaintained since late 2018.

- **rusty-blockparser** [28] is another Bitcoin parser. It that also supports similar blockchains such as Litecoin, Namecoin, and Dogecoin. It converts blockchain data to CSV files that can be used for analyses. The project is unmaintained since May 2018.

- **GraphSense** [29] is a distributed analysis platform for Bitcoin based on Apache Spark and Cassandra. Under the hood GraphSense uses BlockSci for the extract, transform, load (ETL) process. GraphSense is the only domain-specific blockchain analysis tool in this list that is actively maintained. It is developed at the Austrian Institute of Technology, supported by public funding[14]. GraphSense has a web interface to explore blockchain data and supports address clustering. It allows to search for addresses across blockchains, however, it does not support cross-chain address clustering.

- **Neo4j** [30] is a popular general-purpose graph database. Neo4j is used for analyses in [2, 3]. The blockchain2graph project [31] offers a tool that imports the Bitcoin blockchain into a Neo4j database. According to [4], Neo4j is up to 600 times slower than BlockSci due to the lack of domain-specific optimizations.

Of all six presented tools, only GraphSense and Neo4j are actively maintained. GraphSense relies on BlockSci to parse blockchain data, and Neo4j is a general-purpose database with insufficient performance. The rest of the tools are unmaintained and also provide far lower performance than BlockSci. That said, BlockSci is de-facto the only publicly available tool that allows efficient blockchain data analyses of Bitcoin and related blockchains. To the best of our knowledge only GraphSense includes basic optimizations to access data across chains. However, it does not support flexible and programmable cross-chain analyses.

Commercial Tools & Services

Some known vendors are Chainalysis Inc. [32], Elliptic Enterprises Limited [33], and CipherTrace Inc. [34]. All of them specialize in use cases such as forensics, being compliant with regulations ("due diligence"), or performing other security-related investigations. The commercial tools could not be tested for cross-chain support due to their proprietary business model with high licensing fees. According to the information available on the vendor's websites, only Chainalysis Inc. offers some type of multi-currency support: "*across [...] 10 cryptocurrencies you can: [...] build graphs for enhanced due diligence and investigations, understand the counterparties in transactions, track the flow of funds between entities.*" [35]. However, no details about the implementation and its features are given. It also remains unclear whether cross-chain address clustering is supported.

[14] EU's H2020 TITANIUM project and Austrian's FFG VIRTCRIME project

2.3 BlockSci: High-Performance Blockchain Analysis Tool

BlockSci is an open-source blockchain analysis tool developed by Kalodner et al. [4]. Compared to the analysis tools presented in Section 2.2.3, BlockSci provides a powerful, user-friendly interface and very high performance. For example, all Bitcoin transactions can be iterated in a few seconds. This is possible due to a highly optimized, domain-specific architecture. BlockSci converts raw blockchain data into a custom data layout that is optimized for analysis. This data layout is stored on disk and is mapped into memory for analyses. Thus, BlockSci is a domain-specific in-memory database. The analysis library accesses this database and provides an intuitive Python interface. It exposes a `Blockchain` object that represents the entire chain. Using this object the analyst can easily access all data of the blockchain. Listing 2.5 shows code that calculates the average transaction fee for a given month.

Listing 2.5: Sample query to retrieve the average transaction fee in Mar 2019.

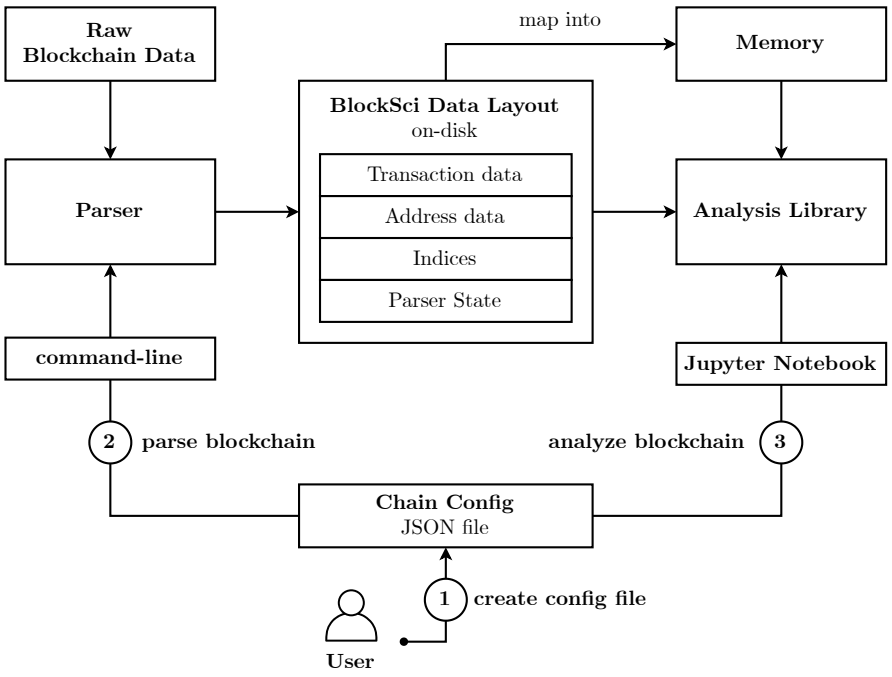
```
import blocksci
chain = blocksci.Blockchain("/btc/config.json")

fees = []
for block in chain.range("Mar 2019"):
    for tx in block:
        fees.append(tx.fee)
print("Avg. tx fee in Mar 2019: " + mean(fees))
```

Besides the Python interface, BlockSci also has a C++ interface that provides higher performance. BlockSci supports blockchains that use the same UTXO-based data format as Bitcoin (Section 2.1.4), e.g., Bitcoin Cash, Bitcoin SV, Litecoin, and ZCash. All of them can be converted to the optimized data layout using BlockSci's parser. Blockchains with a different format are not supported, e.g., Ethereum with its account-model-based layout. BlockSci comes with several useful analysis features. Its address clustering module allows to cluster addresses using the multi-input and the change heuristic. CoinJoin transactions [36] can automatically be detected and excluded from clustering to avoid false positives. BlockSci also includes a currency converter that provides up-to-date exchange rates.

The following sub-sections cover BlockSci's architecture in detail. We only cover the parts of BlockSci that are required to understand the rest of this work. For the parts that are not described we refer to [4] and the official BlockSci documentation [37]. The provided information is valid for the current development version 0.6, in particular Git revision 8681010 in repository [38]. Figure 2.10 gives an overview of BlockSci's architecture. Prior to performing an analysis, the user has to create a JSON-based config file for the blockchain (Step 1). This config file is needed to start the parser (Step 2) that converts raw blockchain data to BlockSci's data layout. Then the user can analyze the blockchain using a Jupyter Notebook that interacts with the analysis library (Step 3). The analysis library maps large parts of the optimized data layout into memory for fast access.

Figure 2.10: Overview of BlockSci's Architecture



2.3.1 Config File

BlockSci's chain config file contains all settings that are needed by the parser and the analysis library. A sample config for Bitcoin is shown in Listing 2.6. The `chainConfig` contains chain-specific settings: `coinName` specifies the name of the coin; `dataDirectory` sets the directory that contains the optimized blockchain data; `pubkeyPrefix`, `scriptPrefix`, and `segwitPrefix` set the P2PK(H), P2SH, and Segregated Witness (SegWit) prefixes needed to generate string representations of addresses; and `segwitActivationHeight` defines the block height of SegWit activation, if activated. The `parser` key configures the parsing process: `coinDirectory` defines the path to the node software's data directory; `blockMagic` defines the chain-specific magic bytes that raw blocks on disk start with; and `hashFuncName` configures the hash function to calculate block hashes, needed to parse blocks in the correct order. The `maxBlockNum` setting defines the block height to parse up to.

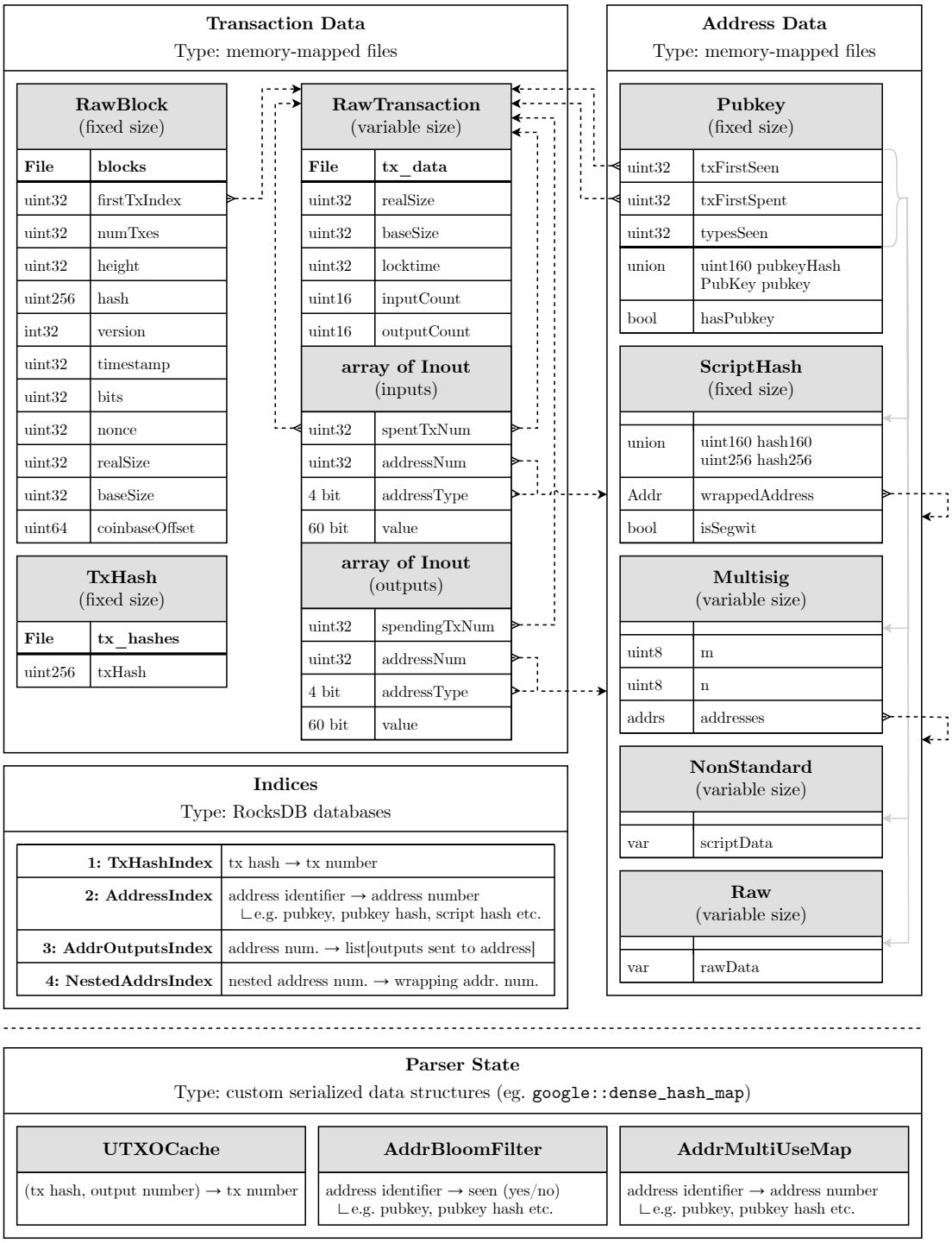
Listing 2.6: Sample BlockSci config file for Bitcoin

```
{
  "chainConfig": {
    "coinName": "bitcoin",
    "dataDirectory": "/parser_output",
    "pubkeyPrefix": [0],
    "scriptPrefix": [5],
    "segwitActivationHeight": 481824,
    "segwitPrefix": "bc"
  },
  "parser": {
    "disk": {
      "coinDirectory": "/bitcoin_node_data",
      "blockMagic": 3652501241,
      "hashFuncName": "doubleSha256"
    },
    "maxBlockNum": 501951
  },
  "version": 5
}
```

2.3.2 Data Layout

After creating the config file the user can start the parser to convert raw blockchain data into BlockSci's optimized data layout. This section covers the data layout, and the next section covers how the parser creates output data corresponding to this layout. Figure 2.11 shows the data layout of BlockSci that we reference to throughout this section. The layout separately stores transaction data, address data, indices, and the parser state. Transaction data and address data are stored as serialized C++ structs, while the indices are stored in a key-value store. The serialized structs allow that the analysis library (Section 2.3.4) can memory-map those files for fast and simple access. The parser state stores data that is needed during the parse process.

Figure 2.11: BlockSci's data layout that is optimized for analysis



Transaction Data

Block metadata (`RawBlock`) and transactions (`RawTransaction`) are stored in separate files. Each transaction struct is followed by a list of inputs and outputs – both are represented by the same struct called `Inout`. Inputs have a reference to the transaction that contains the spent output in `spentTxNum`. Bitcoin's 32-byte transaction hash pointer is replaced with a 4-byte transaction number. Likewise, if an output has been spent, `spendingTxNum` contains a reference to the transaction that contains the spending input. This allows efficient graph traversal, e.g., to follow a flow of coins. Some less frequently accessed data is stored in separate memory-mapped files. For example, only few analyses need to access transaction hashes and thus, they are stored separately (`TxHash`) from the main `RawTransaction` struct. This optimizes memory access and CPU caching by avoiding that not needed data wastes space in cache lines.

Address Data

Address data is stored separately from transaction data. This allows to deduplicate input and output scripts. BlockSci deduplicates scripts in two ways. First, BlockSci has the notion of equivalent addresses [37]. Address types are considered equivalent when the same information is required to spend them, e.g., both P2PK and P2PKH addresses require knowledge of the private key. The data layout only stores a single entry for all equivalent instances of an address. This approach does not only reduce the size of the data structure, but also allows the analysis library to expose "equivalent addresses" to the user. Second, BlockSci deduplicates the scripts itself. Existing address records are reused when the same address occurs multiple times, i.e., when an address receives coins repeatedly.

Every supported address type (`addressType`) has its own struct (`Pubkey`, `ScriptHash` etc.). The serialized structs are stored in separate memory-mapped files per type. BlockSci assigns an incrementing integer (`addressNum`) to every new unique address, e.g., the 10^{th} unique P2SH address gets assigned number 9. This allows direct lookups within the memory-mapped file using the offset calculated by `addressNum * sizeof(ScriptHash)`. Every address in BlockSci can uniquely be identified by the tuple `(addressType, addressNum)`. Inputs and outputs use this tuple to store a reference to the contained address. Each deduplicated address type struct has three common fields that store usage metadata: `txFirstSeen` and `txFirstSpent` reference the transactions where the address first occurred in an output (seen) respectively in an input (spent); and `typesSeen` stores which of the specific equivalent types have been seen, e.g., only P2PK, only P2PKH, or both for `Pubkey`. The usage metadata is followed by individual fields for every address type. For example, given a P2PKH output, the `Pubkey` struct stores the public key hash that is provided in the output script in `pubkeyHash`. As soon as this output is spent by an input, the public key is revealed and BlockSci replaces the hash with the public key in `pubkey`. The hash can still be calculated from the public key, if needed.

Indices

In addition to the memory-mapped transaction and address data structures, BlockSci maintains four indices that are stored in RocksDB databases [39]. RocksDB is a very fast and versatile key-value store that is optimized for SSD storage. It can be optimized for different kinds of workloads and offers many performance tuning options. Note that BlockSci internally uses other, less intuitive index names than given below.

1. **`TxHashIndex`:** *transaction hash* $\rightarrow$ *transaction number*
   This index maps transaction hashes to BlockSci's internal transaction number. It is needed to retrieve transactions by hash.

2. **`AddressIndex`:** *address identifier* $\rightarrow$ *address numbers*
   This index maps address identifiers, e.g., public key, public key hash, or script hash, to BlockSci's internal address number. It is needed to retrieve an address by its address string. The parser uses this index to deduplicate addresses, i.e., to check whether an address has been seen before.

3. **`AddrOutputsIndex`:** *address number* $\xrightarrow{received}$ *list[outputs sent to this address]*
   For every address, this index stores a reference to all outputs that have been sent to this address. It is needed to calculate the balance of an address, among other queries. A separate index is maintained for every supported address type.

4. **`NestedAddrsIndex`:** *nested address number* $\xrightarrow{wrapped\ in}$ *wrapping address number*
   Recall that a P2SH address can wrap another address (Section 2.1.4). BlockSci stores a reference to the wrapped address in `ScriptHash.wrappedAddress`. Additionally, BlockSci stores references from multisig addresses to the contained P2PK(H) addresses in `Multisig.addresses`. This index stores the reverse mapping for both address types. It is needed for BlockSci's equivalent addresses feature. A separate index is maintained for every supported address type.

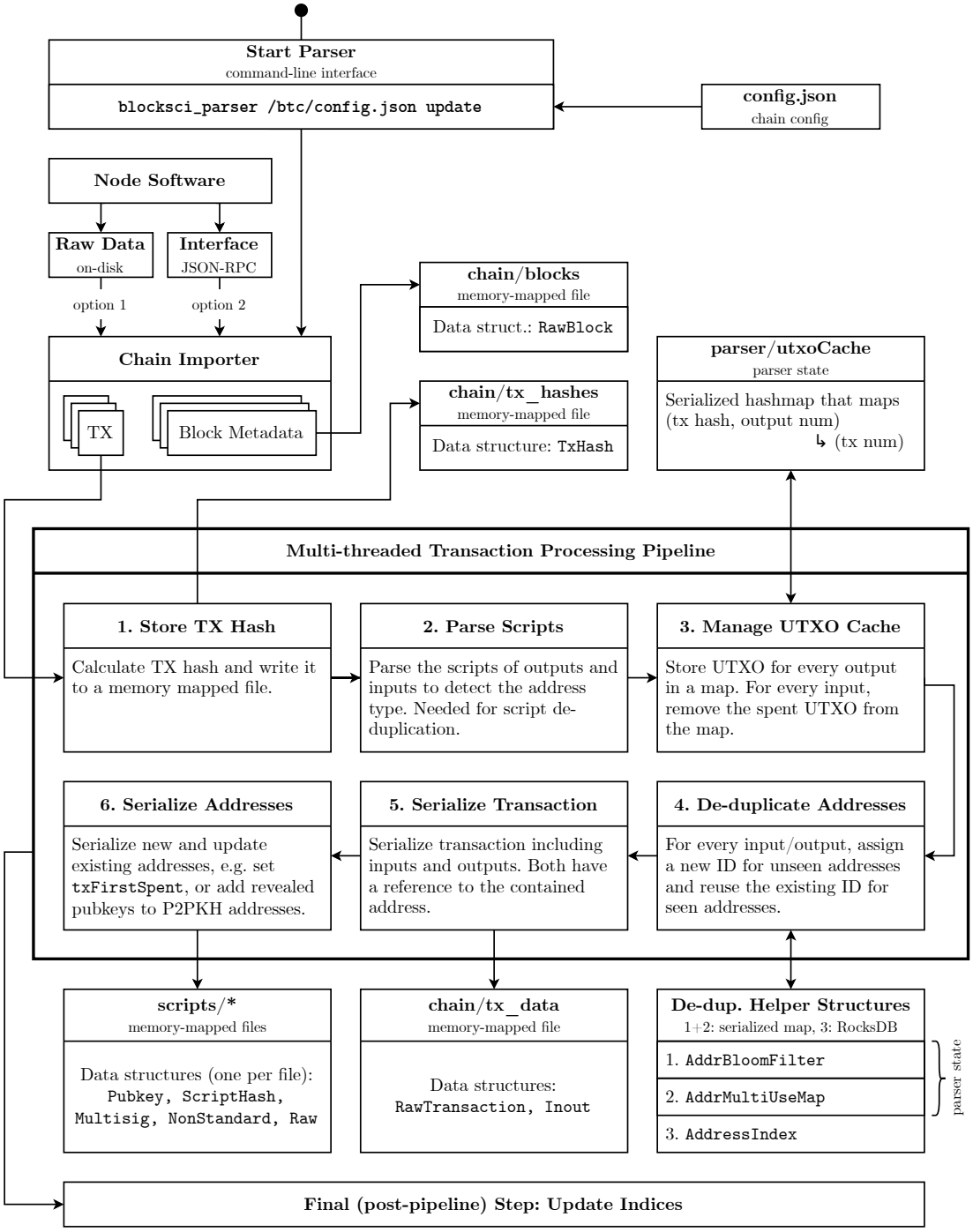
Parser State

Parsing is stateful and the parser needs to maintain the current state between runs. The `UTXOCache` is needed to link new inputs to the output they spend. Both `AddrBloomFilter` and `AddrMultiUseMap` are used for address deduplication. All three data structures are covered in more detail in the next section.

2.3.3 Parser

BlockSci's parser is the component that converts raw blockchain data into the described data layout. This process involves multiple non-trivial and computation-intensive tasks such as script parsing, data deduplication, and linking outputs with the inputs that spend them. This makes the parser the most complex part of BlockSci. Figure 2.12 illustrates the steps performed by the parser. The parser is invoked by the user with the chain config file and the `update` command.

Figure 2.12: Parser sequence to convert raw blockchain data to the optimized layout



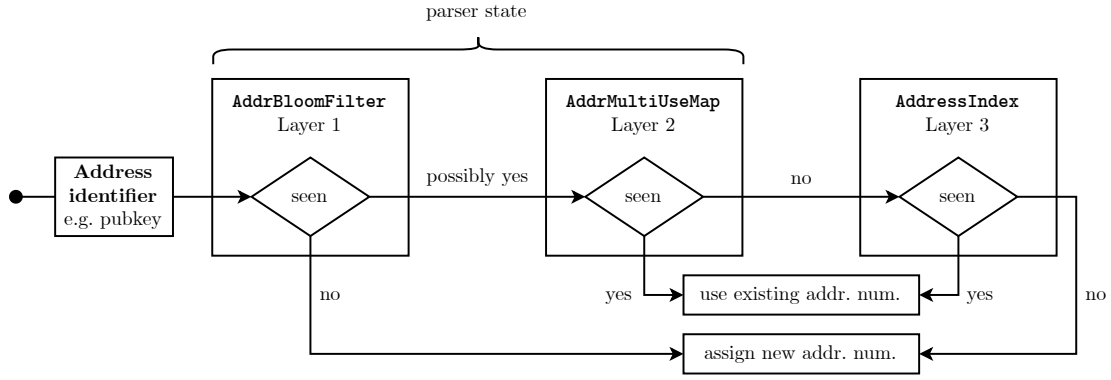
The chain importer supports two input options for raw blockchain data. Data can be read directly from the on-disk node data directory, or via the JSON-RPC interface

that most nodes provide. The first option is very fast due to a node-specific binary reader. The second option is slower but does support more blockchains due to the de-facto standardized JSON-RPC interface. The importer does directly persist block metadata (`RawBlock`) to a memory-mapped file. Transactions need more processing and are passed to a multi-threaded pipeline with six steps[15]. During this process an incrementing integer is assigned to every transaction. A description of every pipeline step is given in Figure 2.12. In the following we cover additional details that are relevant along the pipeline.

As mentioned previously, parsing is stateful. One type of state is the current set of UTXOs (`UTXOCache`). It stores a map from (tx hash, output num) → transaction number and is maintained in Step 3 of the pipeline. The map is needed to link new inputs to the output they spend. An entry is added for every new output. For every input, the entry of the spent output is retrieved and removed from the map. In this step Bitcoin's 32-byte transaction hash pointers are converted to the corresponding transaction number.

Another type of state is needed for address deduplication in Step 4. A required pre-processing task for address deduplication is address type detection by parsing the scripts in Step 2. For the actual deduplication, the parser maintains an index of seen addresses (`AddressIndex`) and a counter for every address type. When the address of a parsed input/output is seen for the first time, a fresh address number is assigned. Otherwise the existing address number is assigned. Resolving whether an address has been seen uses the disk-based `AddressIndex`. For Bitcoin this index has over 40 gigabytes and thus, queries to it are relatively slow. Keeping it in memory would significantly increase the memory requirement. Therefore the parser uses a hierarchical caching mechanism with three layers to improve performance, as shown in Figure 2.13.

Figure 2.13: BlockSci's mechanism to check whether an address has been seen before.



The first layer is a bloom filter that stores all seen addresses. A bloom filter is a probabilistic data structure to check if an element is a member of a set. The results from bloom filters can include false positives, but negative results are always correct. In other words, a bloom filter returns either "certainly not seen" or "possibly seen". Thus, if the

[15]We present a simplified pipeline. BlockSci's actual pipeline has additional (sub-)steps.

bloom filter returns false, i.e., address has not been seen, a new number can be assigned immediately. Otherwise, the second layer is queried: a map that stores all addresses that have previously been used more than once. This is due to the observation that only 8.6% of all Bitcoin addresses are used more than once, but those account for 51% of all occurrences [4]. The last layer is the RocksDB-based `AddressIndex` that maps address identifiers, e.g., a pubkey or pubkey hash, to the address number. The bloom filter and multi-use map are only used by the parser and thus, are stored as part of the parser state. The `AddressIndex`, however, is also used by the analysis library[16] and is stored with the other indices of the optimized data layout. This distinction of storage locations is relevant in Section 3. The described address deduplication is applied to all address types except non-standard and nulldata (`OP_RETURN` outputs), which are always assigned a fresh address number. The result of Step 4 is that every input and output of the processed transaction has an assigned address number.

Another optimization is applied in Step 6 that serializes address data (`Pubkey`, `ScriptHash` etc.) to separate memory-mapped files per address type. For common address types like P2PK(H) or P2SH, the scripts always follow the same pattern of op-codes and data arguments in between (Section 2.1.4). Thus, scripts of one type only differ in the individual data they push onto the stack, e.g., the pubkey, script hash, or signature. To reduce the size of the dataset the parser only stores relevant data for each address type, e.g., the pubkey for P2PK(H) addresses. Common op-codes and signatures do not provide any relevant information and are stripped.

After processing all transactions, the parser updates the RocksDB-based indices `TxHashIndex`, `AddrOutputsIndex`, and `NestedAddrsIndex`. Only the `AddressIndex` is updated as the parser processes transactions.

2.3.4 Analysis Library

Once the parser has converted the blockchain to the optimized data layout, the user can analyze the blockchain using the analysis library. The library is responsible for loading and accessing data. It exposes data via a Python and a C++ interface. The recommended analysis mode is using the Python interface in a Jupyter Notebook. The C++ interface is faster than Python and should be preferred for performance-critical analyses. Using the chain config file the user can initialize a `Blockchain` object, as shown previously in Listing 2.5. This chain object acts as an entry-point to access data of the blockchain. It is iterable so that the user can easily iterate over all blocks using a loop. Blocks are iterable as well to access all contained transactions. The documentation for the full API can be found at [37].

BlockSci uses the same data layout on-disk and in-memory for transaction data and address data. Thus, loading the data simply involves memory-mapping those files using the `mmap()` syscall [40, p. 1017]. Using memory-mapped I/O has several advantages. The operating system can transparently handle many aspects of loading the file: memory management, caching using the page cache, and swapping pages out if the system runs

[16] To retrieve an address by address string, e.g., resolve pubkey hash → address num.

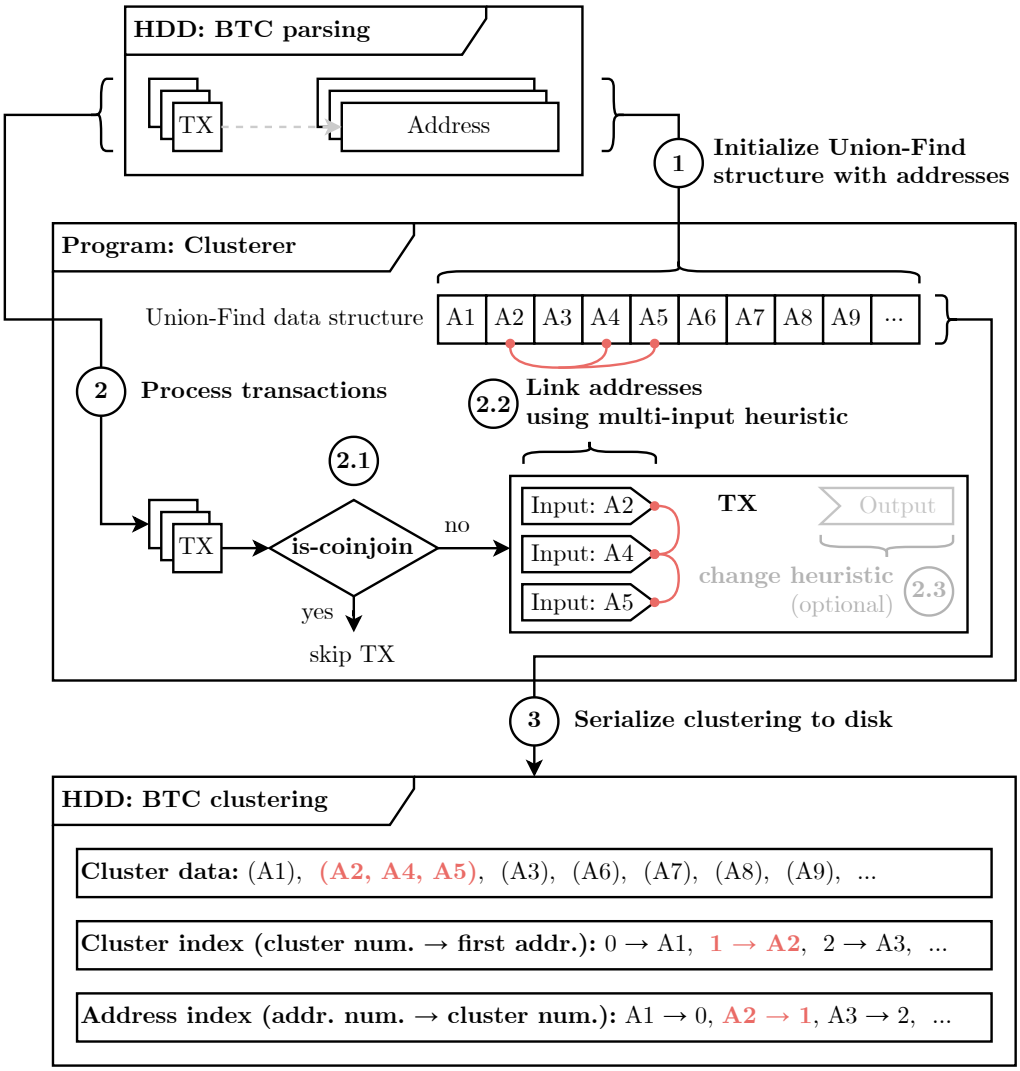
out of memory. No memory needs to be allocated manually, and no objects need to be initialized on the heap or the stack. Instead, once the mapping is created all elements in the file are immediately accessible as C++ structs. The `mmap()` syscall returns a pointer to the beginning of the mapping. Casting this pointer to the type of the underlying data structure, e.g., `RawBlock`, allows to directly address and access elements using the object-oriented interface that the `RawBlock` struct defines. That said, the memory-mapping approach provides instant access to hundreds of million transactions. All data is only loaded into memory on-demand. On first access a page fault occurs and the operating system loads the accessed page into memory. A drawback of memory-mapped I/O is that parser output data is less portable, i.e., it can only be used on the same architecture it was created on. Loading the parsing on other hardware, e.g., with a 32 bit instead of 64 bit CPU, may not work.

Address Clustering

The analysis library comes with a built-in address clustering module[17]. It supports the multi-input and nine different kinds of change heuristics that the user can choose from. Figure 2.14 illustrates BlockSci's address clustering implementation. The first step involves the initialization of a Union-Find data structure with all addresses that appear on the blockchain. A Union-Find or Disjoint-Set data structure allows to partition a set of elements into disjoint subsets. In our case we want to partition addresses into clusters that contain addresses controlled by the same user or entity. BlockSci uses the Union-Find implementation by Jakob [41]. After the Union-Find initialization of Step 1, every address is in its own cluster. In Step 2 all transactions are iterated to apply the clustering heuristics. Substep 2.1 detects and excludes CoinJoin transactions to avoid false positives, i.e., spurious edges in the address graph. The multi-input heuristic is enabled by default and links all addresses that appear in inputs (Step 2.2). An optional change heuristic additionally links the change address with the first input address (Step 2.3). After Step 2, all addresses that are likely controlled by the same entity are linked in the Union-Find data structure. In the final Step 3, BlockSci persists the Union-Find data structure to multiple files on disk. The first file contains all sorted addresses, grouped by cluster, i.e., all addresses of the first cluster are stored after each other, followed by the addresses of the second cluster etc. The second file maintains an index from the cluster number to the offset of the cluster's first address in the first file. The last file maintains the reverse index – from addresses to clusters. Listing 2.7 shows how to create and access an address clustering with BlockSci.

[17] Section 2.2.1 covers address clustering.

Figure 2.14: BlockSci's single-chain clustering



Listing 2.7: Creating and accessing an address clustering using the Python interface

```
import blocksci
chain = blocksci.Blockchain("/btc/config.json")

cluster_manager = blocksci.cluster.ClusterManager.create_clustering(
    "/btc/clustering-output", # clustering output path
    chain                     # chain object to cluster from
)

addr = chain.address_from_string("1G8veZsDpXCVn5m2xg12RKAuWNiJtXYyyw")
cluster = cluster_manager.cluster_with_address(addr)
cluster.address_count() # number of addresses in cluster
```

3 Generalizing BlockSci to Forked Ledgers

In this section we first set the scope and define requirements for the new multi-chain mode. Then we cover all required changes in detail and outline architectural choices. We end with an evaluation of the new mode and a discussion of limitations and future work.

3.1 Requirements

We start by gathering the requirements to improve BlockSci's cross-chain analysis support for forked ledgers. Cross-chain analyses combine the data of multiple chains in a single analysis. This allows to extract new insights which would not be possible by looking at the chains individually. Forked chains provide an effective and powerful data source for cross-chain analyses due to their shared history. We identify two main challenges when performing cross-chain analyses. First, the already high memory requirement of single-chain analyses is further increased with every additional chain. For example, the optimized BTC transaction data (excluding address data) has almost 80 GB as of Dec 2019. Additionally loading BCH's transaction data with 45 GB increases the memory requirement by more than 50% – and therewith also the cost of the analysis. For forked ledgers a large part of this increase may be redundant due to the shared history. In the case of BCH, almost 90% or 40 GB are redundant and only 5 GB constitute relevant new data. Thus, there is great potential to save memory by sharing common data between chains. The second challenge we see is accessing and combining the data of multiple chains efficiently. Cross-chain analyses are most useful when the analysis platform provides means to query the data relationships between chains. Let us look at some examples: given an address, an interesting cross-chain query might be to retrieve the balance of the address on multiple chains. Balance calculation requires to fetch the received outputs of the address per chain, which constitutes another interesting query. Similarly, given a pre-fork UTXO, the analyst might want to query on which chains this output has been spent[1]. This challenge of data retrieval is very broad and it exceeds the scope of this work to implement a comprehensive cross-chain query API.

3.1.1 Scope & Contribution

The overall objective of this work is to improve BlockSci's support for cross-chain analyses. We limit the scope to forked ledgers, as they provide valuable data and allow to significantly optimize the memory footprint. We see our contributions as a first step towards better cross-chain support. In this work we focus on creating a solid foundation

[1] Recall from Section 2.1.3 that pre-fork UTXOs can be spent in both the parent and the forked chain.

that can be improved in the future. We do *not* provide new API methods to access data across chains, like the two example queries given above[2] – this is planned as future work. Instead, we aim to provide the same analysis experience and interface as the current BlockSci version. Every chain is represented via a single object that provides access to data using the existing API. Our main contributions are that 1) addresses are deduplicated, and thus, compatible across chains; and 2) common data between chains is shared in memory. The cross-chain address deduplication allows to utilize the links between forked chains with the existing single-chain API. For example, the analyst can retrieve the same address from both chains and then compare the activity on each chain. Based on this scope we identified the following functional and non-functional requirements for the new multi-chain mode. The key words "MUST", "REQUIRED", "SHOULD", "SHOULD NOT", "MAY", and "OPTIONAL" in Sections 3.1.2 and 3.1.3 are to be interpreted as described in RFC 2119 [42].

3.1.2 Functional Requirements

1. **Normalized addresses:** All addresses MUST be deduplicated across chains. The address representation MUST be compatible across chains. For example, the address with ID 1 MUST correspond to the same underlying address on all chains of a multi-chain parsing. This is an essential requirement for novel cross-chain analyses, such as cross-chain address clustering (Section 4).

2. **Flexible configuration:** The config file MUST allow to define a linear sequence of chains, i.e., one root chain and an arbitrary number of forked chains. The individual fork heights MUST be configurable by the user. For example, a configuration of (Bitcoin $\xleftarrow{478{,}558}$ Bitcoin Cash $\xleftarrow{556{,}766}$ Bitcoin SV) MUST be possible.

3. **BTC and BCH support:** The blockchains Bitcoin and Bitcoin Cash MUST be supported and tested. Adding future support for other blockchains SHOULD be considered when making design choices. Support for chains other than BTC and BCH is OPTIONAL.

4. **Anticipate cross-chain queries:** It is planned to add a cross-chain API in the future. All design choices MUST be made with this goal in mind, i.e., common cross-chain queries SHOULD be considered in the new architecture. For example, indices SHOULD be combined and shared across chains to retrieve data for multiple chains using a single query. Adding new methods to query data cross-chain is OPTIONAL and out of scope.

3.1.3 Non-Functional Requirements

1. **Optimize memory consumption:** The memory footprint of analyses MUST be optimized by sharing data that is common among the configured chains. In particular, identical pre-fork blocks and address data MUST be shared in memory.

[2] An exception are the cross-chain clustering methods introduced in Section 4.

The memory footprint SHOULD be considered for all design choices, e.g., when changing data structures.

2. **Maintain high performance:** BlockSci's high performance SHOULD be maintained to the extend possible, especially in single-chain mode. Also the new multi-chain mode SHOULD offer a similarly high performance.

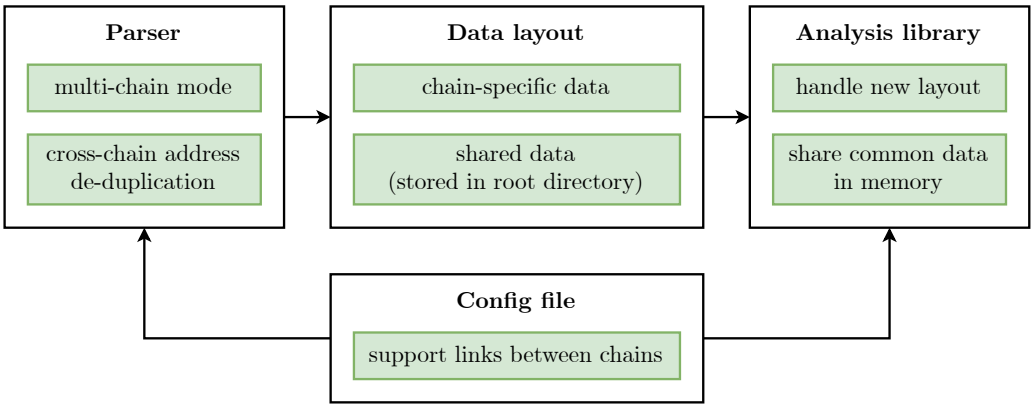
3. **Backwards compatibility:** Changes to the end-user Python interface SHOULD NOT break existing analyses, i.e., backwards compatibility SHOULD be maintained wherever possible. Backwards compatibility of existing parsings is OPTIONAL – a full reparse MAY be needed to use the extended BlockSci version.

4. **Extensibility:** It is planned to add a cross-chain API in the future. All design decisions SHOULD be made with this goal in mind to ensure extensibility. This includes that the already highly complexity of BlockSci SHOULD NOT be significantly increased by the new multi-chain mode. The multi-chain mode SHOULD reuse existing single-chain procedures wherever possible.

3.2 Required Changes

In the following we give a brief overview of the required changes to fulfill the requirements. Figure 3.1 shows BlockSci's architecture (cf. Figure 2.10) and the needed modifications per component. We cover each component in detail in the subsequent sections.

Figure 3.1: Overview of all required changes per component



1. **Config file:** BlockSci uses a separate config file and data directory for every chain. We stick to this design and extend the config file to allow links between multiple forked chains. The data directory of the root chain gets a special role. It stores common data that can be shared across chains, e.g., the deduplicated address data.

2. **Data layout:** BlockSci's existing data layout is optimized for single chains. Several changes to this layout are needed to efficiently store forked chains. We classify

all blockchain data as shared, i.e., valid across chains; or chain-specific, i.e., only valid for a single chain. The new layout reflects this classification to allow sharing data across chains in memory. Other changes are needed where BlockSci's existing fixed-size format does not permit to store data for multiple chains. The `AddrOutputsIndex` RocksDB index also needs modification to store data of multiple chains. The new data layout is used in both single- and multi-chain mode to avoid the maintenance of two separate layouts.

3. **Parser:** A new multi-chain mode is added to the parser. It parses multiple chains using a combination of existing single-chain methods. Reusing single-chain code reduces the required changes and minimizes additional complexity. The new mode handles the deduplication of addresses across chains. The parser also needs adaption to create output according to the new layout in both single- and multi-chain mode.

4. **Analysis library:** The analysis library is changed to correctly load and access the new layout. Identical blocks are loaded only once and the library provides the abstraction of a full chain for each blockchain in a multi-chain configuration.

3.2.1 Config File

The existing BlockSci version uses a separate output directory and config file for every chain (Section 2.3.1). We adhere to this design for the multi-chain mode, but make two modifications. First, the config file needs extension to allow links between chains. We add two new options to the config's `chainConfig` section: `parentChainConfigPath` and `firstForkedBlockHeight`, as shown in Listing 3.1. Forked chains link to the config file of their parent chain via `parentChainConfigPath`. This approach allows an arbitrary number of forked chains to be linked. An empty `parentChainConfigPath` setting denotes the last (=parent or root) chain of a multi-chain config. The `firstForkedBlockHeight` setting specifies the block height of the first forked block, e.g., 478,559 for Bitcoin Cash.

Second, we change the role of the data directory for the root chain. This directory does not only store root chain data, but also data that can be shared across chains. For example, the deduplicated address data for all chains is only stored in the directory of the root chain. For forked chains, the analysis library transparently redirects access to shared data to the directory of the root chain. As a consequence, all data directories of a multi-chain configuration belong together, i.e., a forked chain can not be loaded if the directory of the root chain is not available. The role of the root directory and the mechanism to share data across chains is covered in detail in Sections 3.2.3 and 3.2.4.

Listing 3.1: Chain config file of Bitcoin Cash in a multi-chain configuration

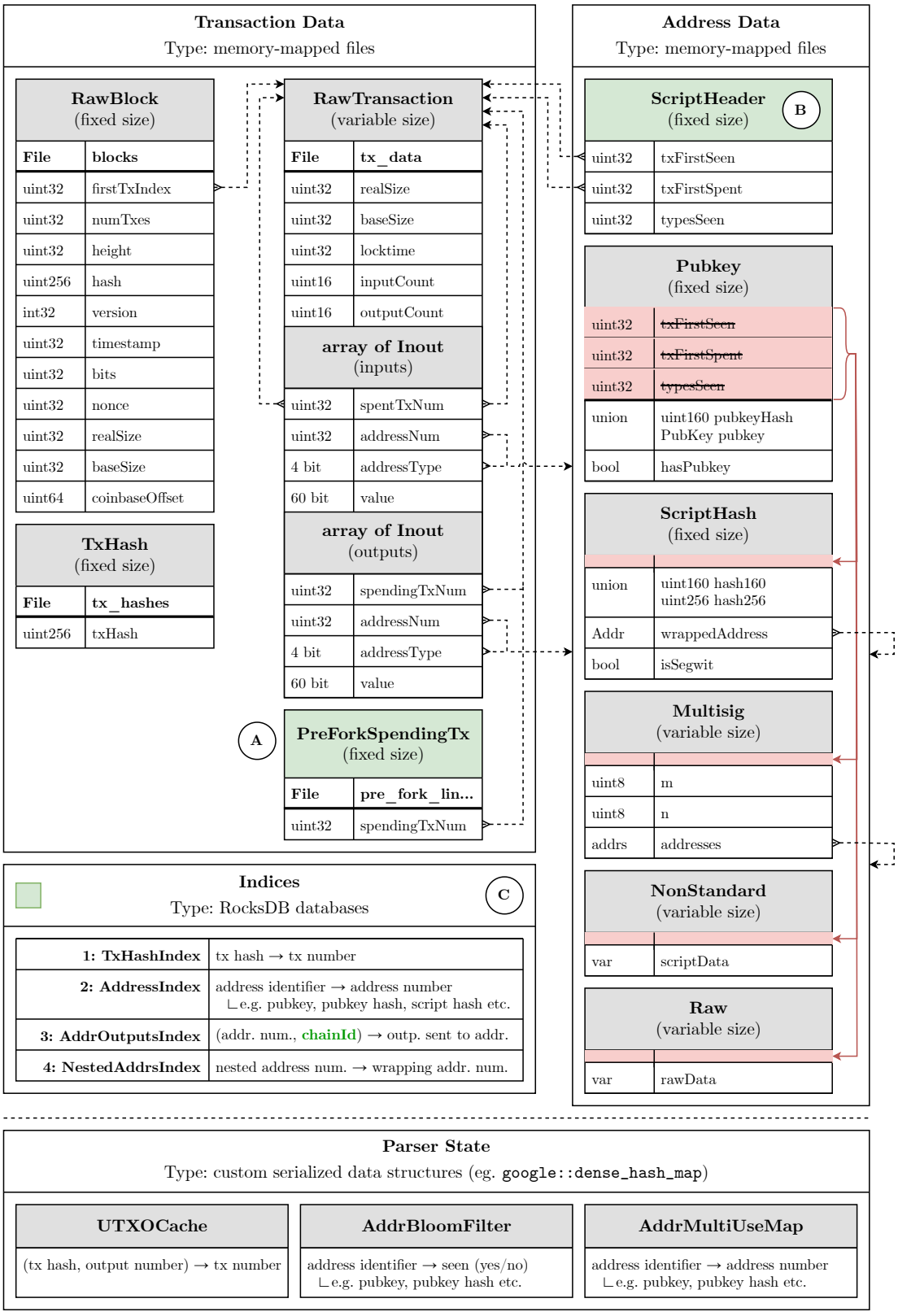
```
{
  "chainConfig": {
    "coinName": "bitcoin_cash",
    "dataDirectory": "/bch/",
    "parentChainConfigPath": "/btc/config.json",
    "firstForkedBlockHeight": 478559,
    "pubkeyPrefix": [0],
    "scriptPrefix": [5],
    "segwitActivationHeight": 481824,
    "segwitPrefix": "bc"
  },
  "parser": {
    "disk": {
      "blockMagic": 3652501241,
      "coinDirectory": "/bitcoin_cash_node_data",
      "hashFuncName": "doubleSha256"
    },
    "maxBlockNum": 501951
  },
  "version": 5
}
```

3.2.2 Data Layout

BlockSci's existing data layout is optimized to store data of a single chain only (Section 2.3.2). A naive solution to support multiple forked chains is to use the same layout and separately store each chain in its entirety. However, this results in a lot of redundant data. It also complicates data sharing: while raw pre-fork blockchain data is identical across forked ledgers, the optimized layout of BlockSci is not. One such instance are links from pre-fork data to post-fork data. For example, the link from every output to the transaction that contains the spending input (`Output.spendingTxNum`). Even though the raw output in the blockchain is identical across chains, this link can legitimately differ between chains, as pre-fork UTXOs can be spent post-fork in both the parent and the forked chain. In the following, we use the term "chain-specific" to refer to such cases where data is only valid for a single chain. The given example is just one instance where the existing data layout is insufficient for the new multi-chain mode.

Thus, we need to change the existing data layout to meet our requirements. It should efficiently store data of multiple forked chains and facilitate sharing data in memory. The new layout should be usable for both single- and multi-chain mode. This allows that the parser can follow a similar logic in both modes to minimize complexity. We make several modifications to the existing data layout (cf. Figure 2.11 in Section 2.3.2). Figure 3.2 shows the updated data layout and highlights all changes. Each modification (A, B, and C) is covered in detail below. A) and B) are cases where chain-specific fields require to hold separate values per chain, but BlockSci's fixed-size structs can only hold values for a single chain. We solve this issue by changing the data layout to store chain-specific fields separately. C) covers the RocksDB-based indices. We add a chain identifier field to the `AddrOutputsIndex` to allow storing data for multiple chains.

Figure 3.2: Required changes to the data layout of BlockSci. The colors represent additions (green) and deletions (red).



A) Outputs: Link to Spending Transaction

BlockSci links every spent output with the transaction that contains the spending input. The fixed-size struct for outputs can only store a single value in the `Output.spendingTxNum` field. However, every pre-fork UTXO can be spent post-fork in both the parent and the forked chain. Each fork thus gets a separate memory-mapped flat file that contains the spending transactions' IDs for all outputs created before the fork – see `PreForkSpendingTx` in Figure 3.2. The parser creates this file during the parse process. The analysis library memory-maps the file and takes care of returning the correct values.

The file stores the spending transaction numbers for *all* pre-fork outputs. For Bitcoin Cash, this corresponds to roughly 2.8 GB of data (4 bytes[3] × 660 million BTC pre-fork outputs). We could optimize and only store the values for "unspent-at-fork-height" outputs. That corresponds to roughly 55 million (8%) of all outputs at the fork height of Bitcoin Cash. This alternative approach requires a data structure that supports fast point queries (by output number), e.g., a hash map. BlockSci uses Google's `dense_hash_map` [43] for serializable hash maps. It has a memory overhead factor of 4 times the size of the contained data. The memory requirement would roughly be 4 bytes × 55 million outputs × 4 overhead factor = 900 MB, i.e., only 30% of the memory-mapped file approach. However, using a hash map has multiple disadvantages. First, elements are in random order instead of contiguous, which results in decreased performance due to worse locality of reference. Second, the hash map is fully loaded into memory (heap) on initialization. It can not be loaded on demand and the operating system can not page out data if it runs out of memory. Third, it is more complex to implement. Thus, we decided to use a memory-mapped file and accept the higher memory requirement.

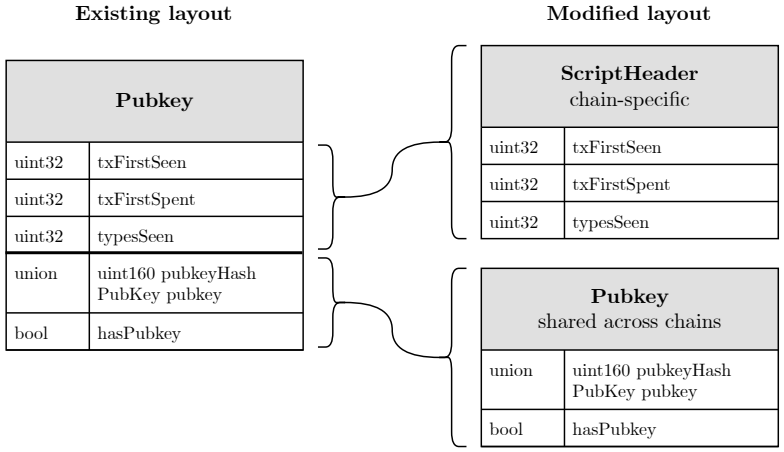
B) Address Metadata

BlockSci supports five common address types and uses a separate struct for each type (`Pubkey`, `ScriptHash` etc.), see Section 2.3.2. Every struct has three common metadata fields that store when the address was first seen (`txFirstSeen`), spent (`txFirstSpent`), and which types were seen (`typesSeen`). Additionally, each struct has individual data per address type, e.g., the public key for P2PK addresses. While the individual data is valid across chains, the metadata fields may differ between chains. For example, an address might be first seen in different transactions on the parent and forked chain.

We solve this issue by splitting all address type structs in two separate structs. The three common metadata fields are moved to a new struct called `ScriptHeader`. It is stored in separate memory-mapped files per type and chain. Figure 3.3 illustrates this modification for the `Pubkey` struct. These changes allow to store and load data that is valid across chains only once – in the directory of the root chain. The chain-specific metadata fields are stored separately for every chain in the data directory of the respective chain. The analysis library manages the access and retrieves metadata from the directory of the chain that is currently accessed.

[3] `sizeof(Output.spendingTxNum)` = 4 bytes

Figure 3.3: All address type structs (here: `Pubkey`) are split in two structs to separately store shared and chain-specific data.



C) RocksDB-based Indices

BlockSci uses four indices that are stored in RocksDB databases, see Section 2.3.2. Three of those indices contain data that is valid across chains. Thus, those indices do not need modification. The `AddrOutputsIndex` stores chain-specific data, but can be modified to hold data for multiple chains. All indices should be shared across chains, i.e., no new database should be created for a fork. This simplifies cross-chain queries as only one index needs to be queried instead of separate indices per chain. In the following we discuss every index to determine required changes.

1. **`TxHashIndex`:** *transaction hash* → *transaction number*
   This index maps transaction hashes to BlockSci's internal transaction number. Intuitively this index should be valid across chains, as transaction hashes ought to be unique across all blockchains. However, replay attacks can cause that transactions with the same hash appear on different chains [44]. The attacker takes a valid transaction of the parent chain and replays it on the forked chain, or vice versa. In either way, the spent outputs must be unspent on both chains or the replayed transaction will be invalid. Given that the forked chain does not implement replay protection, the result are transactions with the same hash on both chains. According to the functional requirements in Section 3.1.2, we must support Bitcoin and Bitcoin Cash. As Bitcoin Cash implemented replay protection from the beginning, we do not change this index [10, Req. 6-2].

   However, the index should be changed in the future to support chains without replay protection, e.g., Bitcoin SV. This can be done by adding a chain ID field after the transaction hash to store mappings for multiple chains. The resulting index can be queried for a single chain (by adding the chain ID to the query). Additionally, it also supports queries across chains with a prefix-based range query

using the transaction hash (without the chain ID). This is useful to find transaction duplicates, i.e., replayed transactions. We leave this change for future work.

2. **`AddressIndex:`** *address identifier* → *address numbers*
   This index maps address identifiers, e.g., public keys, to BlockSci's internal address number. Address identifiers like public keys are valid across chains. The same public key can be used on multiple chains, e.g., in P2PK outputs, and it does not change. Also the hash of the public key, e.g., in P2PKH outputs, does not change given that the same hash function is used. A fork that changes the hash function would complicate address deduplication and require significant changes to the multi-chain mode. However, we are not aware of a relevant fork that changed the hash function. Address numbers are also valid across chains due to the deduplication of the new multi-chain parse mode (Section 3.2.3). Thus, no modifications to the index are needed to store data for multiple chains. The index can also be shared across chains without any changes.

3. **`AddrOutputsIndex:`** *(address num., chain ID)* $\xrightarrow{recv.}$ *list[outputs sent to address]*
   For every address, this index stores a reference to all outputs that have been sent to this address. The data of this index is chain-specific: an address may receive different sets of outputs on each chain. Thus, the index needs modification to be shared across chains. We add a 1-byte chain ID field after the address number to store entries for multiple chains. A new `ChainId enum` contains pre-defined IDs for common chains, see Section 3.2.4. This allows point queries for outputs of an address on a specific chain. It also allows prefix-based range queries to retrieve the outputs of an address on all chains (by omitting the chain ID).

4. **`NestedAddrsIndex:`** *nested address number* $\xrightarrow{wrapped\ in}$ *wrapping address number*
   For every P2SH address BlockSci stores a link to the wrapped address. Similarly, every multisig address has links to the contained P2PK(H) addresses. This index stores the reverse mapping. The new multi-chain parse mode deduplicates addresses across chains (Section Section 3.2.3). Thus, the stored address numbers are valid across chains and no modifications to the index are needed.

3.2.3 Parser

The parser is the most complex component of BlockSci. It converts raw blockchain data into the optimized data layout using a multi-threaded processing pipeline (Section 2.3.3). The existing parser creates independent parsings and processes one chain at a time only. Several changes are needed to jointly parse multiple chains. According to our requirements we try to keep the required changes to a minimum. This is to prevent that the already complex parser becomes even harder to maintain. We identified three core changes that are covered in detail in the following. First, the parser needs modification to support the new config file. Second, as the data layout has changed, the parser needs to create outputs according to the new layout. Third, a major modification is the addition of a multi-chain parse mode that can parse related chains together.

Updated Config File

As a first step the parser is updated to support the new config file format that allows links between chains. It is implemented as a linked list of chain configurations. Each chain configuration links to its parent configuration. The implementation allows to distinguish between the root and forked chains, and to get the paths of their data directories. As mentioned in Section 3.2.1, a separate data directory is used for every chain. The directory of the root chain is used to also store data that is shared across chains. The roles of the directories are discussed in more detail below.

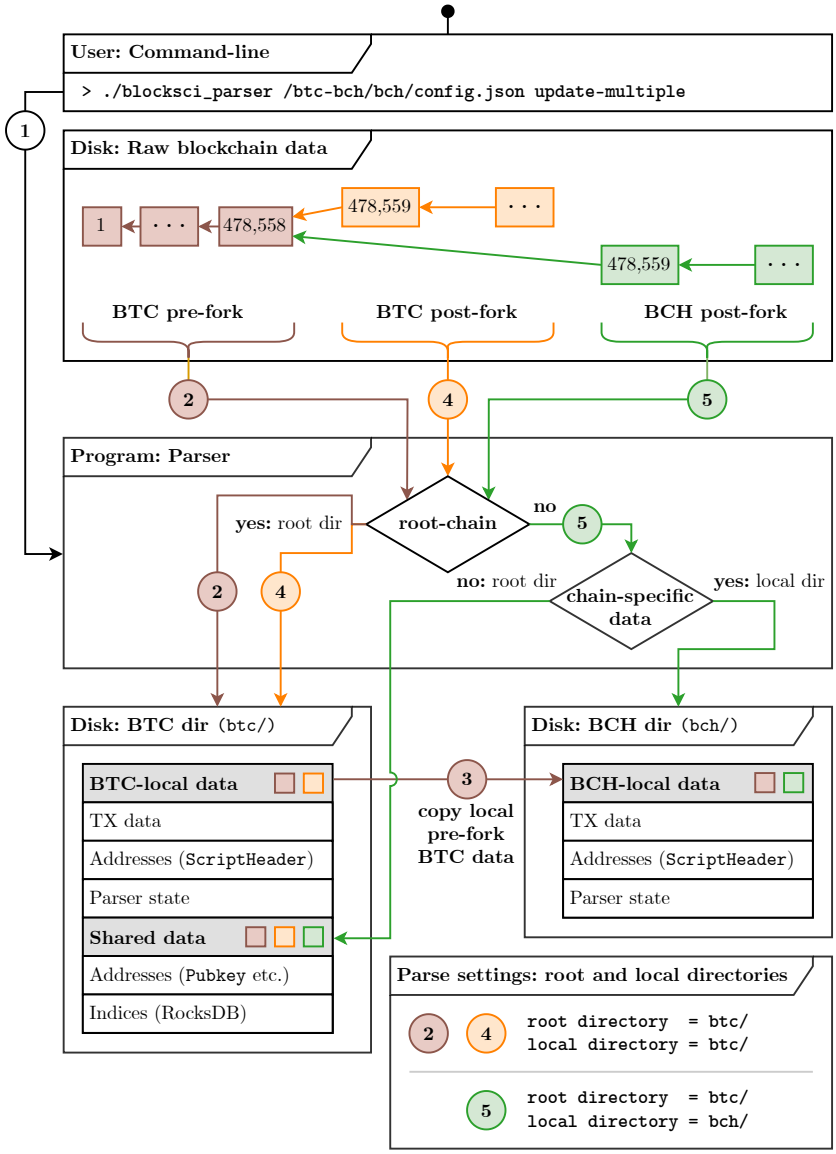
Updated Data Layout

As Section 3.2.2 covers all changes to the layout in detail, we only briefly cover some implementation details here. It is important to understand that the updated layout is the new default for both parse modes, i.e., the new layout fully replaces the old layout. For example, the new file containing spending tx numbers for every output (see change A in Section 3.2.2) is created in both parse modes, even though it is not used in single-chain mode. Similarly, the chain-specific `AddrOutputsIndex` does have the chain ID field also in single-chain mode. This adds a small storage and processing overhead, but allows that a similar parser and analysis library logic can be used for both modes.

Multi-Chain Mode

The main modification of the parser is the addition of a new multi-chain parse mode. Our strategy is to implement this new mode by reusing and combining the existing single-chain logic. This way we avoid an entirely separate new mode with dedicated logic that needs maintenance. We start with the fact that the existing parser works incrementally. It can parse a chain up to a given block height, and continue parsing to a higher height later. We make use of this feature for the new multi-chain mode, as illustrated step by step in Figure 3.4. The idea is that the root chain, e.g., BTC, is parsed first (Step 2) but only up to the fork height of its forked chain, e.g., BCH. The resulting output is a valid parsing for BTC and BCH, as their raw data is identical up to the fork height. Thus, we can safely copy the parser output of BTC's data directory into BCH's data directory (Step 3). The parser can then sequentially continue to process chains. First BTC is parsed up to the latest block height (Step 4), followed by BCH (Step 5). All parsings are executed using the existing single-chain logic. This works because all data including the parser state is copied in Step 3. Thus, when parsing BCH in Step 5 the parser "assumes" it is continuing a previous parsing. The result are two independent parsings of BTC and BCH up to the given block heights – something that could be created with the existing BlockSci version as well.

Figure 3.4: The sequence of the parser's new multi-chain mode.



The missing piece is a mechanism to share and combine data between chains, e.g., to deduplicate addresses across chains. This is where the previously mentioned new roles of root and forked data directories come to play. We classify all parser output data structures as either shared, i.e., valid across chains; or chain-specific, i.e., not valid across chains. Below we discuss the classification for relevant data structures. Based on this classification the parser stores data in either the root directory or the local directory, see Table 3.1. Then the parser applies a simple rule to its output: shared data (and chain-specific data of the root chain) is stored in the root directory, and chain-specific data of forks is stored in the respective directory of the fork. This is done by assigning

a root and local directory to every chain that is parsed, as shown in the bottom-right box in Figure 3.4. For example, when BCH is parsed in multi-chain mode (Step 5), the root path is BTC's directory and the local path is BCH's directory. The root chain is an exception and has identical root and local directories. Implementing the described rule is straightforward: using the updated config file format of Section 3.2.1, each parsed chain has access to the root data directory and its own local directory. BlockSci's config classes provide methods to get the paths of the subdirectories that hold different types of data, e.g., `chain/` for transaction data, `scripts/` for address data etc. We change those methods to return either the root or the local path, depending on the type and classification of data.

This design solves the issue of sharing data between chains during the parse process. First, shared data of a multi-chain configuration is written to the root chain's directory. Second, and more importantly, the parser also *reads* shared data from this directory when it continues parsing a chain incrementally (Steps 4 and 5 in Figure 3.4). For example, the address index (`AddressIndex`) is shared and thus, the same database is opened for every chain of a multi-chain configuration. This inherently solves address deduplication across chains. For all addresses that are seen – e.g., on BTC in Step 2 – an entry is added to the address index. The entry represents a mapping from address identifier (e.g., the pubkey) to the internally assigned address number. When the same address is encountered on BCH in Step 5, the previously assigned address number is retrieved from the shared address index. The described design also allows that the multi-chain mode is largely based upon the existing single-chain parse logic. Modifications are only needed at the top level to orchestrate the process, as shown step-by-step in Figure 3.4. Additionally, the configuration classes need to be changed to return the correct directory based on the type of data.

Classification of Data: Shared or Chain-Specific

Below we discuss which types of data are shared across chains. This determines where the parser stores the data. Shared data is stored in the root chain's directory. Chain-specific data is only valid for a single chain and is stored in the local directory. Note that the classification is only relevant for the described mechanism that the parser uses for data sharing. The analysis library also respects this classification, but implements additional measures to improve data sharing, e.g., for identical pre-fork data.

1. **Transaction data** is generally chain-specific. Although pre-fork transaction data is identical on both chains, it is currently duplicated in every fork's data directory. This is a consequence of copying the parent to the forked directory in Step 3 of Figure 3.4. Step 3 is needed so that the existing single-chain mode can be used to continue parsing with post-fork BCH data in Step 5. However, the data is only duplicated on disk and optimizing storage consumption is not a requirement, as disk space is cheap. The analysis library takes care of loading common pre-fork transaction data into memory only once.

2. **Address data** is both shared and chain-specific, depending on the fields. The

metadata fields in `ScriptHeader` that store information about the address' usage are chain-specific. The address-type-specific data structures (`Pubkey`, `ScriptHash` etc.) can be shared across chains.

3. All **indices** can be shared across chains as they can hold data for multiple chains after the modification described in B) of Section 3.2.2.

4. For the **parser state** data sharing in memory is less important from a performance perspective because it is only used during the parse process, but not for analyses. The `UTXOCache` contains the current set of UTXOs. It must not be shared, as it would break the parse process when pre-fork UTXOs are spent in both the parent and the forked chain[4]. The `AddrMultiUseMap` and `AddrBloomFilter` are used for address deduplication. Both could technically be shared, but we decided to not share `AddrMultiUseMap` as we speculate the map might defeat its purpose when addresses satisfy "multi-use" because they are used once in both chains. We leave the evaluation of sharing parser state for future work.

Table 3.1: Storage location for parser output data based on the data type

Data	Data structures[1]	Classification	Directory
TX data	`RawBlock`, `RawTransaction`, `TxHash`, `PreForkSpendingTx`	chain-specific[2]	local
Addresses	`ScriptHeader`	chain-specific	local
	`Pubkey`, `ScriptHash`, `Multisig`, `NonStandard`, `Raw`	shared	root
Indices	`TxHashIndex`, `AddressIndex`, `AddrOutputsIndex`, `NestedAddrsIndex`	shared	root
Parser state	`UTXOCache`, `AddrMultiUseMap`	chain-specific	local
	`AddrBloomFilter`	shared	root

[1]: As shown in Figure 3.2

[2]: Only relevant for the parser, the analysis library shares pre-fork data in memory.

3.2.4 Analysis Library

The analysis library is responsible for loading existing parsings and provides the end-user interface to query data (Section 2.3.4). It manages the data access to all types of data: transaction data, address data, and indices. Several modifications are required to support the new multi-chain mode. First, the library needs changes to correctly access the new data layout and share common data in memory. We discuss the required changes for

[4]The first spend removes the UTXO from the `UTXOCache`, and the query for the second spend fails.

different types of data. Second, the end-user API needs to be changed so that the chain ID and name can be retrieved for every object. These API changes do not affect backwards-compatibility.

Data Access

The new multi-chain parser mode stores shared and chain-specific data in separate directories (Section 3.2.3). The analysis library uses a similar mechanism as the parser to access data. A root directory and a local directory is assigned to every chain that is loaded. The library then uses the same rule as the parser: shared data is retrieved from the root directory, and chain-specific from the local directory. This mechanism can be applied to single-chain parsings as well by setting the root and local directories to the same path. While this approach already solves some data access issues, more specific changes are required based on the type of data.

1. **Transaction data:** Blocks, transactions, inputs, and outputs are generally chain-specific, except for common pre-fork data, which can be shared with the parent chain. In-memory data sharing is implemented by transparently redirecting access to pre-fork data to the parent directory. While we speak of a directory here, remember that transaction data files are memory-mapped and thus, loaded into memory. Thus, it is equally correct that all requests to pre-fork blocks and transactions are redirected to the memory region of the parent chain. The analysis library knows the fork block height from the config file and can derive the number of the last pre-fork transaction. This information is used to redirect access based on the requested block height respectively transaction number. As inputs and outputs are accessed via the containing transaction, access to them is automatically redirected as well. However, outputs need special treatment due to their chain-specific link to the spending transaction and the therefore updated data layout, as described in A) in Section 3.2.2. When an output object is instantiated, it is checked whether the output was created before the fork. If so, the `spendingTxNum` value is retrieved from the separate fork-specific `PreForkSpendingTx` file, while the rest of parent chain's output data is also valid for the forked chain. Otherwise, the output data is read from the local directory of the fork, which contains the correct chain-specific `spendingTxNum` value (as it is post-fork data).

2. **Address data:** In the new layout every address has chain-specific metadata (`ScriptHeader`), and an individual struct per address type that is shared across chains (`Pubkey` etc.). The analysis library is changed so that the chain-specific `ScriptHeader` is always loaded from the local directory. The individual address type structs are loaded from the root directory. This ensures that correct data is retrieved based on the chain object that the request originates from.

3. **Indices:** The RocksDB-based indices can all be shared due to the changes described in C) in Section 3.2.2. Thus, the parser stores them in the root directory. The analysis library redirects access accordingly using the modified configuration classes.

API Changes

Section 3.1.1 limits the scope of the multi-chain mode to providing the user with an abstraction of separate single chains using the existing single-chain API. Therefore no additional data retrieval methods are added to the end-user API. The existing BlockSci API does not allow to retrieve to which chain an object, e.g., an output, belongs. The analyst has to mentally keep track of from which chain object data was retrieved. Therefore we add a `chain_id` property to all objects to help the user distinguishing between chains. The property returns the same ID that is used by the parser in the new data layout. The chain IDs are implemented as an `enum` with predefined 1-byte values for every chain and their variants, e.g., `BITCOIN=1`, `BITCOIN_TESTNET=2` etc.

3.3 Usage

Installation

The code of this work can be found on GitHub at `mplattner/BlockSci` [38] in the branch `feature/fork-support`, revision a178437. Installation and compilation instructions are provided in the official BlockSci documentation [45].

Parser

Listings 3.2 and 3.3 show a sample multi-chain configuration for BTC and BCH. We assume that both chains have their individual data directory at `/btc` respectively `/bch`. The BCH config contains a link to the BTC config and specifies the fork height.

Listing 3.2: BTC config `/btc/cfg.json`

```
{
  "chainConfig": {
    "coinName": "bitcoin",
    "dataDirectory": "/btc",


    "pubkeyPrefix": [0],
    "scriptPrefix": [5],
    "segwitActivationHeight": 481824,
    "segwitPrefix": "bc"
  },
  "parser": {
    "disk": {
      "blockMagic": 3652501241,
      "coinDirectory": "/node/btc",
      "hashFuncName": "doubleSha256"
    },
    "maxBlockNum": 610696
  },
  "version": 5
}
```

Listing 3.3: BCH config `/bch/cfg.json`

```
{
  "chainConfig": {
    "coinName": "bitcoin_cash",
    "dataDirectory": "/bch",
    "parentChainConfigPath": "/btc/cfg.json",
    "firstForkedBlockHeight": 478559,
    "pubkeyPrefix": [0],
    "scriptPrefix": [5],
    "segwitActivationHeight": 2147483647,
    "segwitPrefix": "NONE"
  },
  "parser": {
    "disk": {
      "blockMagic": 3652501241,
      "coinDirectory": "/node/bch",
      "hashFuncName": "doubleSha256"
    },
    "maxBlockNum": 615796
  },
  "version": 5
}
```

Listing 3.4 shows how to start the parser in multi-chain mode using the new `update-multiple` command. The parser expects that the provided config file is from the leaf chain, i.e., the deepest fork in the multi-chain configuration. In our example this is BCH. The output messages reflect the multi-chain mode parsing sequence that we outline in Figure 3.4: first BTC is parsed up to the fork height, then the output data is copied to BCH's directory, and in a last step the parser continues to parse BTC up to the specified height, followed by BCH.

Listing 3.4: Creating a parsing using the new multi-chain mode

```
> blocksci_parser /bch/cfg.json update-multiple

-- Performing initial multi-chain parse

- Parsing bitcoin up to block height 478558
Adding 478558 blocks: done
Updating indices: done
Copying data to fork directory: done

- Parsing bitcoin up to block height 610696
Starting with chain of 478558 blocks
Adding 132137 blocks: done
Updating indices: done

- Parsing bitcoin_cash up to block height 615796
Starting with chain of 478558 blocks
Adding 137237 blocks: done
Updating indices: done
```

Analysis Library

The scope of this work is to provide users the known single-chain experience with a backwards-compatible interface. Thus, the usage of the analysis library has not changed with the introduction of the new multi-chain mode. Listing 3.5 shows how users can individually load BTC and BCH of the created multi-chain parsing. This is done in the same way as in the non-extended version: by instantiating a `Blockchain` object with the path to the chain's config file. The analysis library transparently handles sharing data in memory. While both chains are loaded in this example, it is possible to only load a single chain or subset of chains from a multi-chain parsing. The analysis interface is backwards-compatible, with one minor exception described in Section 3.4.3. The documentation for the full API can be found at [45].

Listing 3.5: Loading the chains of a multi-chain parsing using the Python interface

```
import blocksci

btc = blocksci.Blockchain("/btc/cfg.json")
bch = blocksci.Blockchain("/bch/cfg.json")

for btcBlock in btc:
  for btcTx in btcBlock:
    # do something with the tx

for btcBlock in btc:
  for btcTx in btcBlock:
    # do something with the tx
```

3.4 Evaluation

In this section we evaluate the correctness, performance, and backwards compatibility of the created extension. For all measurements, we use Git revision 8681010 for the non-extended version, and revision a178437 for the extended version, both in repository [38]. Both versions are compiled with GCC 7.5 using CMake's release build, i.e., with most compiler optimizations turned on. All parsings are up to BTC block height 610,696 and BCH block height 615,796, corresponding to Dec 31, 2019.

3.4.1 Correctness

We evaluate the correctness of the created extension by comparing the output data of BlockSci with and without the extension. Figure 3.5 shows the setup of the correctness evaluation. First, a version of the parser without the extension is used to create parsings for Bitcoin and Bitcoin Cash. We make the assumption that this parser version – prior to our changes – creates correct results. Second, we use another version of BlockSci that includes the extension. We create parsings for Bitcoin and Bitcoin Cash in single-chain mode, and one parsing for both chains combined using the new multi-chain mode. Third, we use a custom integrity check program to compare the results. It is not possible to compare the parser output files directly because they include struct padding, which is not initialized by default and thus, contains random data. Instead, the integrity checker loads the parsed chain and iterates all transactions. It calls all data retrieval methods of blocks, transactions, inputs, outputs, and the addresses referenced by them. The return values of these methods are sequentially fed into a hash function. The result is a single SHA256 hash that uniquely identifies the parser output data. We compare the (non-extended and extended version) hashes of the same chain to evaluate if the results haven't changed and thus, are correct. Table 3.2 shows the results of this process. The hashes of the non-extended and the extended version match for both BTC and BCH. This indicates that the extended BlockSci version produces correct results.

Figure 3.5: Correctness evaluation setup. Solid lines represent the creation of parsings, and dashed the comparisons of those.

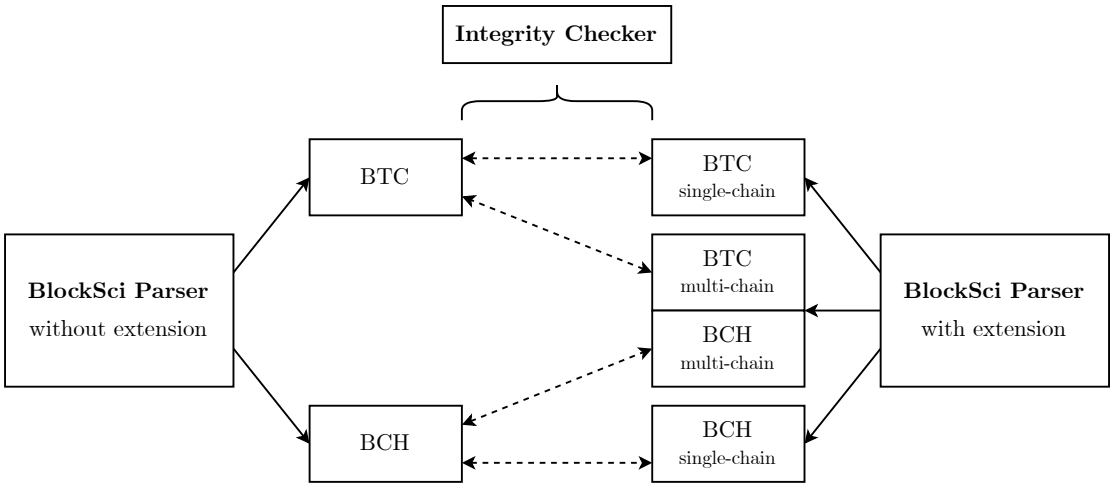


Table 3.2: Result of the integrity check to evaluate correctness.

Chain[1]	**Version**	**Parse Mode**	**Hash**[2]	**Result**
BTC	w/o extension	single-chain	`1798777da0...5497a681da`	*reference*
	w/ extension	single-chain	`1798777da0...5497a681da`	✓
		multi-chain	`1798777da0...5497a681da`	✓
BCH	w/o extension	single-chain	`605d6a1897...6242fd0fa2`	*reference*
	w/ extension	single-chain	`605d6a1897...6242fd0fa2`	✓
		multi-chain	`605d6a1897...6242fd0fa2`	✓

[1]: `maxBlockNum` setting: 610,696 for BTC, 615,796 for BCH, corresponding to Dec 31, 2019
[2]: `Transactions (API w/ address strings)` output of the integrity checker

A limitation of the correctness evaluation is that not all parser output data is included. We do not include all data of the individual structs per address type (`Pubkey` etc.). Their data might legitimately differ between single- and multi-chain parsings. For example, the public key of a P2PKH address might be revealed due a spend on BTC, but not on BCH. As public keys are shared across chains in multi-chain mode, the public key is also available when loading BCH from a BTC-BCH multi-chain parse. On the other hand, a single-chain BCH parsing does not include the revealed public key for the given P2PKH address. In consequence, comparing hashes of a BCH single-chain and a BCH multi-chain parsing would result in erroneously reported inconsistencies. Thus, for addresses we only include the chain-specific `ScriptHeader` data, and the address string. Including the address string instead of the raw public key or public key hash mitigates above

limitation. The address string does not change regardless of whether the public key of the address has been revealed. Thus, it reflects whether the stored data (public key hash *or* public key) is correct. This issue causes that some address data is not hashed by the integrity checker: `Pubkey.hasPubkey`, `ScriptHash.{wrappedAddress,isSegwit}`, `Multisig.addresses`, `NonStandard.scriptData`, and `Raw.rawData`.

3.4.2 Runtime Performance

According to the non-functional requirements in Section 3.1 the high performance of BlockSci should be maintained to the extend possible. We evaluate this requirement with several benchmarks on the analysis library and the parser. All measurements are performed on a bare-metal machine with a quad-core CPU, 64 GB of memory, and a high-performance SSD[5]. The used operating system is Ubuntu 18.04, which is installed on a separate SSD. We flush the operating system's page cache before starting benchmarks[6].

Analysis Library

The performance of the analysis library is evaluated by running various sample queries with the extended and the non-extended version of BlockSci. We use eight tests that implement common real-world queries. Table 3.3 gives an overview of the tests. The upper part shows which data structures every tests iterates, together with the iteration order. The listings below provide Python-like pseudocode[7] for every test. The tests are evaluated for the same six parsings as shown in Figure 3.5 for the correctness evaluation. Table 3.4 shows the execution time in wall clock time averaged over 10 runs for all relevant combinations of parsings.

[5]CPU: Intel Xeon E3-1275 v5 with 3.6 GHz, RAM: 64 GB DDR4-2400 with ECC, Disk: local NVMe SSD with 3.84 TB and read speeds up to 3000 MB/s

[6]Using this command: `sync && echo 3 > /proc/sys/vm/drop_caches`

[7]The C++ code of all tests is available in the `benchmark/` folder of the Git revisions given in Section 3.4.

Table 3.3: Tests to evaluate the runtime of the analysis library

		Test 1: maxInput	Test 2: maxOutput	Test 3: maxFee	Test 4: maxFeeRand	Test 5: nonZeroLocktime	Test 6: nonzeroLocktimeRand	Test 7: zeroConfOutput	Test 8: addrReceivedValue
Accessed data	`Block`	●	●	●		●		●	
	`Transaction`	●	●	●	●	●	●	●	
	`Input`	●		●	●				
	`Output`		●	●	●			●	●
	`Address`								●
	`AddressIndex`[1]								●
Order	Sequential	●	●	●		●		●	●[1]
	Random[2]				●		●		
	Graph traversal[3]							●	

[1]: RocksDB index, [2]: access TXes in random order, [3]: via `Output.spending_tx`

Test 1: maxInput

```
# find maximum input value
cmax = 0
for block in chain:
  for tx in block:
    for input in tx.inputs:
      cmax = max(cmax, input.value)
```

Test 2: maxOutput

```
# find maximum output value
cmax = 0
for block in chain:
  for tx in block:
    for output in tx.outputs:
      cmax = max(cmax, output.value)
```

Test 3: maxFee

```
# find maximum fee
cmax = 0
for block in chain:
  for tx in block:
    cmax = max(cmax, tx.fee)
```

Test 4: maxFeeRand

```
# same as Test 3, but random tx traversal
cmax = 0
for tx_num in random_tx_nums:
  tx = Transaction(tx_num)
  cmax = max(cmax, tx.fee)
```

Test 5: nonZeroLocktime

```
# count txes with locktime > 0
count = 0
for block in chain:
  for tx in block:
    count += (tx.locktime > 0)
```

Test 6: nonzeroLocktimeRand

```
# same as Test 5, but random tx traversal
count = 0
for tx_num in random_tx_nums:
  tx = Transaction(tx_num)
  count += (tx.locktime > 0)
```

Test 7: zeroConfOutput

```
# count outputs that are created and
# spent in the same block
count = 0
for block in chain:
  for tx in block:
    for output in tx.outputs:
      if output.is_spent:
        s_tx = output.spending_tx
        if s_tx.height == block.height:
          count += 1
```

Test 8: addrReceivedValue

```
# calculate received value of a
# gambling service address
address = Address(
  "1dice97ECuByXAvqXpaYzSaQuPVvrtmz6"
)
total = 0
for output in address.outputs:
  total += output.value
```

Table 3.4: Results of the runtime evaluation for the analysis library

Test	**w/o extension**		**w/ extension**		**Change**[4] **(%)**		
	Parsing[1]	**Mean**[2]	**Parsing**[1,3]	**Mean**[2]	**Parsing**	**Chain**	**Test**
1. maxInput	BTC	7.587	BTC sc	7.545	-0.55%	-0.49%	-0.47%
			BTC mc	7.554	-0.43%		
	BCH	4.346	BCH sc	4.327	-0.43%	-0.46%	
			BCH mc	4.325	-0.48%		
2. maxOutput	BTC	8.039	BTC sc	7.831	-2.59%	-2.54%	-2.56%
			BTC mc	7.839	-2.49%		
	BCH	4.634	BCH sc	4.515	-2.57%	-2.57%	
			BCH mc	4.514	-2.58%		
3. maxFee	BTC	9.347	BTC sc	10.641	13.84%	13.97%	14.63%
			BTC mc	10.664	14.09%		
	BCH	5.313	BCH sc	6.121	15.20%	15.29%	
			BCH mc	6.131	15.38%		
4. maxFeeRand	BTC	311.834	BTC sc	329.115	5.54%	5.91%	8.35%
			BTC mc	331.405	6.28%		
	BCH	174.729	BCH sc	188.147	7.68%	10.80%	
			BCH mc	199.050	13.92%		
5. nonz.Locktime	BTC	5.110	BTC sc	5.156	0.91%	0.11%	-0.21%
			BTC mc	5.075	-0.69%		
	BCH	2.992	BCH sc	2.972	-0.68%	-0.53%	
			BCH mc	2.980	-0.39%		
6. nonz.Lockt.Rand	BTC	244.627	BTC sc	268.264	9.66%	9.99%	17.43%
			BTC mc	269.862	10.32%		
	BCH	137.857	BCH sc	153.804	11.57%	24.87%	
			BCH mc	190.472	38.17%		
7. zeroConfOutput	BTC	151.647	BTC sc	166.336	9.69%	9.92%	14.42%
			BTC mc	167.032	10.15%		
	BCH	80.044	BCH sc	88.135	10.11%	18.92%	
			BCH mc	102.245	27.74%		
8. addrReceivedVal.	BTC	0.387	BTC sc	0.398	2.98%	2.60%	3.37%
			BTC mc	0.395	2.22%		
	BCH	0.382	BCH sc	0.402	5.19%	4.15%	
			BCH mc	0.394	3.10%		
Average change[5]	BTC	–	BTC sc	–	**4.93%**	**4.93%**	**6.87%**
			BTC mc	–	**4.93%**		
	BCH	–	BCH sc	–	**5.76%**	**8.81%**	
			BCH mc	–	**11.86%**		

[1]: `maxBlockNum` setting: 610,696 for BTC, 615,796 for BCH, corresponding to Dec 31, 2019

[2]: Average execution time of 10 runs in wall clock seconds, [3]: sc/mc = single-chain/multi-chain parsing

[4]: Relative change w/o ext. to w/ ext.: per parsing, avg. over chain (sc and mc), avg. over test

[5]: Average change over all tests

We observe an average slowdown of 6.87% over all parsings and tests. The tests `maxInput`, `maxOutput`, `maxFee`, and `nonzeroLocktime` implement the common use-case of sequentially iterating a subset of (blocks, transactions, inputs, and outputs). All but `maxFee` run marginally faster with the extended version: `maxInput` (-0.47%), `maxOutput` (-2.56%), and `nonzeroLocktime` (-0.21%). The `maxFee` test is 14.63% slower with the extended version. At the time of writing we have no explanation for this significant slowdown. After all, this test is largely a combination of `maxInput` and `maxOutput`. We plan to further investigate and profile this test in the future.

We observe two more tests that are on average significantly slower with the extended version: `nonzeroLocktimeRand` (17.43%), and `zeroConfOutput` (14.42%). The `nonzeroLocktimeRand` test randomly accesses transactions by transaction number. In the extended version this requires to check whether the number is pre- or post-fork in order to redirect access to the correct memory location (Section 3.2.4). We conjecture that this check causes a large portion of the observed slowdown. For the BCH multi-chain parsing we observe the largest slowdown (38.17%), as all pre-fork transactions are actually redirected to the parent chain. This check also causes `maxFeeRand` to be 8.35% slower on average (13.92% for BCH multi-chain). Sequential iterations of transactions (using blocks) are not affected by this check as then it is only executed once per block.

The `zeroConfOutput` test is 14.42% slower on average. It sequentially accesses all transaction outputs and retrieves the spending transaction for every output., i.e., the transaction graph is traversed. Such graph traversal queries are expected to be slower with the extended version. This is due to the additional check whether the spending transaction number needs to be fetched from the separate pre-fork file (Section 3.2.2).

Overall, we observe an average slowdown of 4.93% and 5.76% for single-chain BTC respectively BCH parsings. This largely fulfills our requirement to maintain the high single-chain performance of BlockSci. However, we see several opportunities for improvement. For example, some of the multi-chain mode checks can be skipped in single-chain mode. We plan to perform additional benchmarks and optimizations in the future.

Parser

The performance of the parser is evaluated by creating multiple parsings and measuring the runtime. We create the same parsings as shown for the correctness evaluation, see Figure 3.5. Table 3.5 shows the average runtime over 3 executions in wall clock minutes.

Table 3.5: Results of the runtime evaluation for the parser

without extension		**with extension**		**Change(%)**
Parsing[1]	**Runtime**[2]	**Parsing**[1]	**Runtime**[2]	
BTC	228.7	BTC single-chain	285.5	24.85%
BCH	81.7	BCH single-chain	104.8	28.33%
BTC/BCH[3]	310.3	BTC/BCH multi-c.	356.3	14.80%
				avg. 22.66%

[1]: `maxBlockNum` setting: 610,696 for BTC, 615,796 for BCH, corresp. Dec 31, 2019

[2]: Average execution time of 3 runs in wall clock minutes

[3]: Accumulated time to parse BTC and BCH: $228.7 + 81.7$

Using the extended version we can observe an increase in parser runtime of 22.66% on average. We conjecture that this is due to the changed data layout that is used in both single- and multi-chain mode. The extended parser uses the same logic in both modes and thus, also performs multi-chain mode tasks for single-chain parsings. However, we do not consider this slowdown a major limitation. After performing the initial parse, updating a parsing with new blocks is near-instantaneous. We strongly recommend using SSD storage as we saw an up to ten-fold increase in parse runtime with HDD storage.

3.4.3 Backwards Compatibility

According to the non-functional requirements in Section 3.1, the end-user Python API should remain backwards compatible, i.e., our changes should not break existing code. At the same time, we limited the scope to providing the existing single-chain API to the user, without adding or changing methods for cross-chain data retrieval. Given this scope, maintaining backwards compatibility is straightforward.

Only a single method in `Address` needs an updated return type due to the cross-chain address deduplication. Recall that every address has a link (`txFirstSeen`) to the transaction where the address was first seen. BlockSci exposes this link to the user via `Address.first_tx` to get the referenced transaction. The new data layout stores this link separately for every chain, as time of first address usage may differ across chains (see B in Section 3.2.2). An address may also be *unseen* on one of the chains in a multi-chain configuration. In this case `Address.first_tx` can not return a `Tx` object. Thus, the return type of this method needs to be changed from `Tx` to `Optional[Tx]`, i.e., it can also return Python's `None` type. This change does not break existing analyses as long as the code is used with a single-chain parsing. A single-chain parsing inherently has a `txFirstSeen` value for every address. Existing code that uses `Address.first_tx` and is executed on a multi-chain parsing will require minor changes. That said, the Python API of the extended BlockSci version should be fully backwards compatible with existing analyses.

3.5 Discussion

In Section 3 we covered the extension of BlockSci to better support cross-chain analyses of forked ledgers. We added a new multi-chain mode that allows to jointly parse and analyze forked chains. Common data between chains is shared in memory and addresses are deduplicated across chains. In Section 3.4.1 we evaluated the result for correctness by comparing the outputs of the extended and non-extended version. The results indicate that the extension works correctly. The extension is a valuable contribution to BlockSci and provides a solid foundation for cross-chain analyses that can be improved in the future.

Fulfillment of Requirements

In Section 3.1.2 we defined four functional requirements (FR) and four non-functional requirements (NFR). In the following we briefly discuss the fulfillment of these requirements. We are aware that the specification of some requirements is not precise enough to rigorously assess the fulfillment. For example, it is difficult to validate if "*all design choices were made with [some goal] in mind*" (FR 4, NFR 1, NFR 4) – only we can (subjectively) answer this question. Other requirements like maintaining the high performance of BlockSci (NFR 2), or maintaining backwards compatibility (NFR 3) were evaluated in Section 3.4. That said, in Table 3.6 we rate the fulfillment of every requirement to the best of our knowledge. Below we discuss requirements that are difficult to verify objectively.

Table 3.6: Fulfillment of the functional and non-functional requirements in Section 3.1

Type	No.	Requirement	Priority	Result
Functional requirements	1.	Normalized addresses	MUST	✓[1]
	2.	Flexible configuration	MUST	✓[1]
	3.	BTC and BCH support	MUST	✓[1]
	4.	Anticipate cross-chain queries	MUST	✓[3]
Non-functional requirements	1.	Optimize memory consumption	MUST	✓[3]
	2.	Maintain high performance	SHOULD	✓[2]
	3.	Backwards compatibility	SHOULD	✓[2]
	4.	Extensibility	SHOULD	✓[3]

[1]: can be verified by using the new feature, [2]: was verified in Sections 3.4.2 and 3.4.3
[3]: is difficult to verify objectively and thus, is discussed below

1. **FR 4:** We anticipated several cross-chain queries when making design choices. The design of the `AddrOutputsIndex` index allows to query the received outputs of an address for a specific chain, for multiple, or for all chains. This allows many interesting cross-chain queries, e.g., calculating the balance of an addresses on

multiple chains. The `TxHashIndex` is planned to be extended in the future so that duplicate transactions across chains can be retrieved. The design of the link from outputs to the spending transaction allows to retrieve all transactions across chains that spend a pre-fork output.

2. **NFR 1:** We did optimize the design for a low memory footprint. Identical pre-fork data is only loaded into memory once and is shared between chains. Address data is deduplicated and also shared in memory. Thus, when loading multiple chains the potential amount of data in memory is significantly lower. The analysis library heavily uses memory-mapped I/O to load files, allowing the operating system to swap memory pages at any time. BlockSci can thus run on machines with insufficient memory, albeit with significantly decreased performance. This makes it difficult to reliably measure the difference in memory consumption across versions. We conjecture the runtime might be a useful proxy that can indicate extensive swapping.

3. **NFR 4:** We made design choices with the consideration that a cross-chain API will be added in the future (also see FR 4 above). We tried not to significantly increase the complexity of BlockSci by reusing existing methods wherever possible. This worked well for the new multi-chain mode of the parser, which is largely based on existing single-chain code.

Limitations

BlockSci supports Bitcoin and blockchains that use a similar format. These chains can have minor differences, e.g., in the consensus rules, without causing incompatibilities with BlockSci. However, we found an interesting edge-case that is currently not supported by the new data layout. It affects Bitcoin and Bitcoin Cash when they are parsed together in multi-chain mode. It causes the Bitcoin parsing to be incorrect for a negligible number of pre-fork P2SH addresses. Recall that P2SH addresses can wrap any other address type (Section 2.1.4). On Bitcoin there exist some pre-fork UTXOs that wrap a Segregated Witness (SegWit) address. We have not covered SegWit so far, but for now it is enough to know that it is a Bitcoin enhancement (BIP141 [46]) that changes the transaction structure and adds new address types, e.g., Pay-to-*Witness*-Public-Key-Hash (P2WPKH). SegWit was activated in Bitcoin at August 23, 2017, i.e., *after* the Bitcoin Cash fork. We conjecture the observed pre-fork P2SH-wraps-P2WPKH outputs were created by developers for testing purposes. Due to the nature of the SegWit implementation, on Bitcoin Cash these outputs can be spent by anyone who knows the redeem script that matches the hash in the P2SH output. The redeem script is revealed when the pre-fork P2SH-wraps-P2WPKH output is spent on Bitcoin. Then anyone can take the redeem script and spend the output on Bitcoin Cash using a non-standard script. The parser detects the wrapped address on Bitcoin as P2WPKH, but as non-standard on Bitcoin Cash, i.e., as two distinct addresses – which is technically correct. However, the `P2SH` struct is shared across chains and can only store one link to the wrapped address in `P2SH.wrappedAddress`. Currently this causes the `wrappedAddress` value to

be overwritten when the non-standard BCH input is parsed. We found that this bug affects 16 pre-fork `P2SH` addresses, i.e., $6 \times 10^{-7}\%$ of all 27 million pre-fork P2SH addresses. This is the number of pre-fork P2SH-wraps-P2WPKH addresses that have been spent on BTC as of Dec 2019. More addresses might be affected that have not been spent yet and thus, can not be detected. The bug did not affect the correctness evaluation because the `P2SH.wrappedAddress` field is not included in the evaluation for reasons described in Section 3.4.1. There are several options to fix or workaround this bug. However, a proper fix needs to store the links to all wrapped addresses, and possibly introduce individual address equivalence classes per chain. This is non-trivial and requires significant changes to BlockSci's data layout. We leave this fix for future work and assign low priority given the marginal number of affected addresses.

Future Work

We see our contributions as a first step towards efficient and user-friendly cross-chain analyses with BlockSci. Thus, we see several opportunities for future work. First, we propose to implement a cross-chain API to query data across chains. This would allow users to more conveniently extract information from the relationships between forked ledgers. The challenge is to design an API that is user-friendly and compatible with the current architecture. We suggest methods like `Address.balance(chain_id)` and `Address.outputs(chain_id)` for addresses, and `Output.spending_tx(chain_id)` for outputs are a useful addition to the current single-chain API. Second, we see potential to further optimize the performance of our multi-chain extension. In Section 3.4.2 we showed that some benchmarks run significantly slower on multi-chain parsings. We propose to profile the current implementation to identify bottlenecks and inefficiencies. For example, the analysis library currently performs some checks in both modes although they are only required in multi-chain mode. Third, the `TxHashIndex` should be adapted to support multi-chain configurations with chains that do not implement replay protection. Parsing such chains, e.g., Bitcoin SV, currently overwrites the existing (tx hash $\rightarrow$ tx number) entry for every duplicate transaction, resulting in a corrupt parsing. We propose to add a chain ID field to the index so that it can store entries with the same transaction hash across multiple chains.

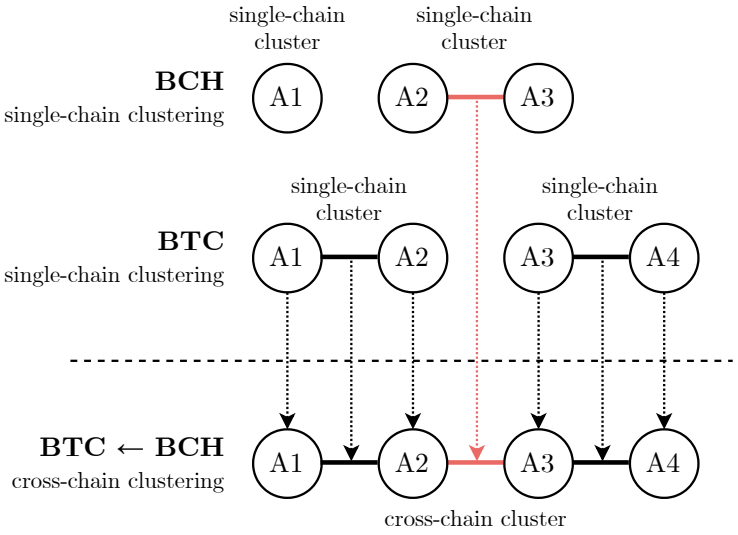
4 Application: Cross-Chain Address Clustering

This section covers the implementation of a novel address clustering technique using the extended BlockSci version of Section 3. First we present the idea of cross-chain address clustering. Then we cover its implementation using BlockSci. We end with the analysis of a cross-chain clustering of Bitcoin and Bitcoin Cash.

4.1 Introduction

In blockchain systems it is trivial to generate new addresses, and thus, identities. Many wallets take advantage of this ability and create a new address for every transaction. The motivation is to protect the privacy of the user, as reusing addresses allows to filter for transactions of a specific user. Address clustering is an established technique that uses heuristics to link addresses that are controlled by the same user (Section 2.2.1). It is used for research, e.g., to understand trends or evaluate privacy; but also in practice, e.g., for cryptocurrency forensics and criminal investigations. BlockSci's new multi-chain mode of Section 3 enables memory-efficient and user-friendly analyses of data across a parent chain and its forks. We utilize the new multi-chain mode to implement a novel clustering technique that includes data of multiple chains: cross-chain address clustering. The technique uses the known heuristics: multi-input and change. These heuristics are applied on the transactions of multiple chains to create an enhanced clustering for the target chain. Figure 4.1 illustrates this new technique. The two upper graphs show single-chain clusterings of BTC and BCH, created using the established technique described in Section 2.2.1. Cross-chain clustering merges these single-chain clusterings to create an improved clustering. Improved means that the address graph has more edges and thus, more links between addresses. For example, the edge between A2 and A3 is only found in BCH and can be propagated to BTC. Adding this edge has significant effects on the clustering: it results in the collapse of BTC's single-chain clusters (A1, A2) and (A3, A4) to a single cross-chain cluster containing four addresses. The propagated edge does not only link A2 and A3, but also all addresses of the affected single-chain clusters: (A1, A3), (A2, A4), and (A1, A4). These three links could not have been found using Bitcoin or Bitcoin Cash alone. More formally, when two clusters C_1 and C_2 with $|C_1|$ and $|C_2|$ addresses are merged, $|C_1| \times |C_2|$ edges are implicitly added to the graph. Thus, every edge added using cross-chain clustering can potentially have large effects on the resulting clustering and the privacy of users.

Figure 4.1: Two single-chain clusters on the BTC blockchain are merged into a cross-chain cluster based on the link between A2 and A3 found in BCH.

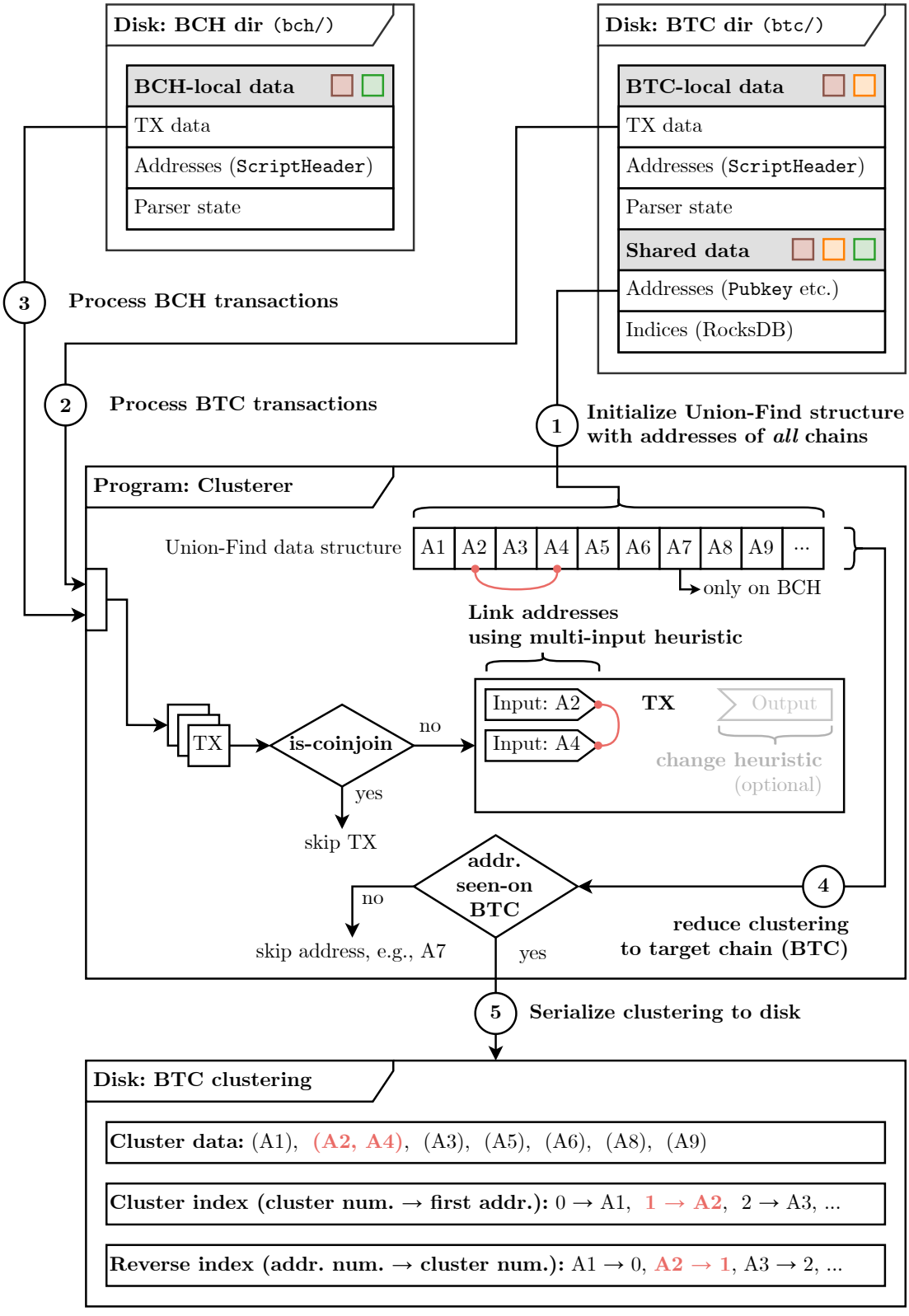


Forked chains are a powerful clustering data source as both chains share a common history. Thus, users who held coins before the fork automatically own coins on the forked chain too. User behavior on forked chains, however, may differ widely: a generally privacy-conscious user who carefully crafts transactions on one chain may perform privacy-harming transactions on another, for example to cash-out all coins. In fact, privacy-harming behavior has already been observed across forks of the Monero blockchain [24]. Cross-chain clustering utilizes that addresses common to multiple forked chains may be spent differently in order to identify additional links between address clusters. [6]

4.2 Implementation

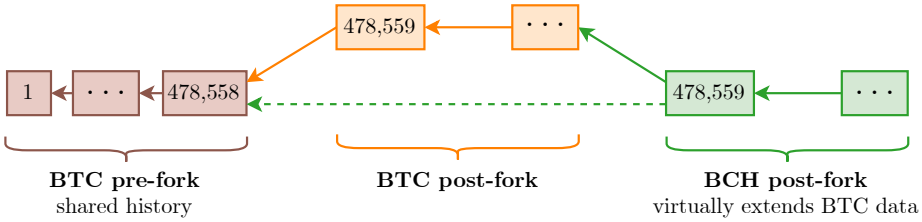
The non-extended BlockSci version already comes with a single-chain clustering module, as described in Section 2.3.4. We want to extend this module to support cross-chain clustering. A core requirement for cross-chain address clustering is that addresses are deduplicated across chains, i.e., inputs and outputs of all chains must refer to the same addresses database. For example, the address identified by (P2PKH, ID 1) must correspond to the same underlying address in all chains that should be clustered. The new multi-chain mode of Section 3 supports this requirement: it deduplicates address data across chains and stores it in a single database in the directory of the root chain. Thus, the implementation of cross-chain clustering is straightforward, as shown in Figure 4.2 for a BTC-BCH clustering. As a first step, the multi-chain-aware clustering module initializes the Union-Find structure with the addresses of *all* chains. Due to the shared address database the addresses can be retrieved with a single lookup.

Figure 4.2: Implementation of cross-chain clustering using the new multi-chain mode.



Then the clustering module sequentially applies the heuristics to all chains. In our example, it first clusters BTC (Step 2), followed by BCH (Step 3). Each additional chain increases the number of transactions that are used for the clustering. Another way to think about it is that the additional chain virtually extends the main chain, as shown in Figure 4.3.

Figure 4.3: Post-fork BCH virtually extends the BTC chain with useful clustering data.



Thus, the actual clustering logic can remain the same for the single- and cross-chain clustering module. After processing the transactions of both chains, the Union-Find structure contains a clustering that includes addresses and links for both chains. Finally, Steps 4 and 5 are responsible for serializing the clustering to disk. While Step 5 is equal in single- and cross-chain mode, Step 4 performs necessary pre-processing in cross-chain mode. Step 4 reduces the clustering to addresses that appear on the user-defined target chain. All addresses that do not appear on the specified target chain are removed from the clustering. This check can be performed via addresses' `ScriptHeader.txFirstSeen` property that stores when an address has first been seen on each chain, see B) in Section 3.2.2. The target chain can be any of the chains that are clustered. This step is required because the scope of the extended BlockSci version is to provide the known single-chain interface, see Section 3.1.1. Having clusterings that contain addresses that do not exist on the main chain would break the existing interface.

4.3 Usage

We add a new `create_clustering_multichain` method to BlockSci's `ClusterManager` class. It allows the user to create a cross-chain clustering given a list of chains, the target chain, and an output directory. Listing 4.1 shows the usage of this new method.

Listing 4.1: Create cross-chain clustering for BTC and BCH, reduce to BTC.

```
# import modules
import blocksci
from blocksci.cluster import ClusterManager

# load chains
btc = blocksci.Blockchain("/btc/cfg.json")
bch = blocksci.Blockchain("/bch/cfg.json")

# create cross-chain clustering
ccClusteringBtc = ClusterManager.create_clustering_multichain(
    "/clustering",             # clustering output path
    [btc, bch],                # chains to cluster, root chain first
    blocksci.chain_id.bitcoin  # target chain (to reduce to)
)
```

Listing 4.2 shows the output of Listing 4.1. The output messages reflect the cross-chain clustering sequence that we outline in Figure 4.2: first the Union-Find structure is initialized, then the Bitcoin transactions are processed, followed by the Bitcoin Cash transactions; then the clustering is reduced to the target chain and serialized to disk.

Listing 4.2: Output of Listing 4.1 to create a cross-chain clustering.

```
Creating clustering based on 2 chain(s), reducing result to addresses seen in bitcoin

Preparing data structure for clustering: done

Clustering using bitcoin data (610696 blocks): done
Clustering using bitcoin_cash data (615796 blocks): done

Post-processing: resolving cluster nums for every address : done
Post-processing: remapping cluster IDs and reducing to bitcoin: done
Reduce result: excluded 43272428 out of 969971931 addresses (4.46%)

Saving cluster data to files
Finished clustering with 926699503 addresses in 561877635 clusters
```

4.4 Analysis: BTC-BCH Cross-Chain Clustering

4.4.1 Preliminaries

Terminology

Cross-chain address clustering is a new technique. Thus, we briefly introduce the terminology we use throughout this section. A *clustering* is a graph of all unique addresses of a blockchain. Every connected component of the graph is a *cluster* that represents the addresses of one real-world entity[1], i.e., a cluster contains one or more addresses. A *merge* links two addresses based on heuristics, which results in the collapse/merge of the two

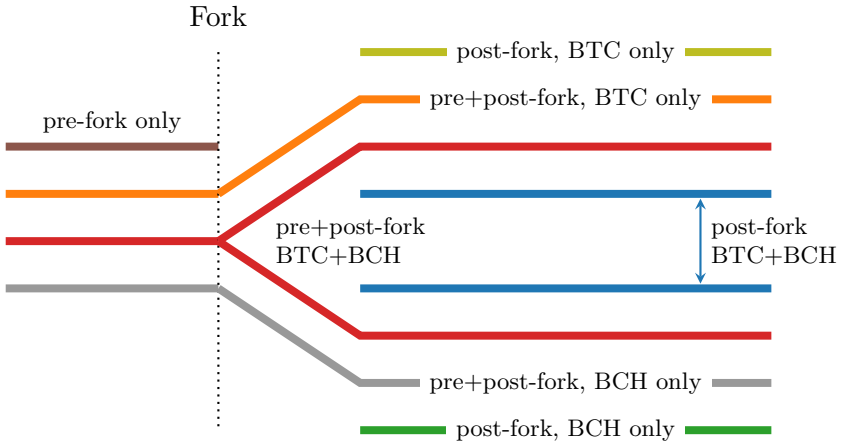
[1]Note that address clustering is heuristic and thus, contains false positives and negatives.

containing clusters. Given two clusters C_1 and C_2, a merge of any two addresses $A_1 \in C_1$ and $A_2 \in C_2$ has the same effect: the two clusters collapse. A clustering created with data of one chain[2] is a *single-chain clustering*. We use *single-chain cluster* to refer to a cluster of a single-chain clustering. A clustering created with data of multiple chains[3] is a *cross-chain clustering*. We assume that each cross-chain clustering is reduced to a single *target chain* before analysis, i.e., the clustering only contains addresses that appear on the target chain. We use Bitcoin as the target chain and Bitcoin Cash as the additional *auxiliary chain* throughout this section. Given a single- and a cross-chain clustering for the same (target) chain, one can analyze which single-chain clusters collapsed in the cross-chain clustering. Every such collapse represents one *cross-chain merge*, i.e., a merge that a) has an effect in the resulting cross-chain clustering, and b) can not be found via single-chain clustering. When two or more single-chain clusters are merged by cross-chain merges, the result is a *cross-chain cluster*, i.e., every cross-chain cluster contains multiple single-chain clusters.

Address Usage Across Forks

Each address may be used on different parts of the parent and/or forked chain: pre-fork, post-fork, or both. We systematize the usage types of addresses across forked chains on the example of Bitcoin and Bitcoin Cash in Figure 4.4.

Figure 4.4: Address usage types on forked chains [6]



Addresses that held coins before the fork may continue to be used on either (orange or gray), or both chains (red). Orange addresses represent users that only touched their pre-fork coins on post-fork BTC, while gray represents the same for post-fork BCH. New addresses may be used after the fork on either chain (yellow or green), or start to appear on both chains despite no pre-fork use (blue). These blue addresses indicate an

[2]For example, Bitcoin.
[3]For example, Bitcoin and Bitcoin Cash.

interesting user behavior that we discuss in Section 4.4.4. Addresses may also cease to see use after the fork (brown). [6, in almost verbatim form]

The address usage types help to clarify the effectiveness of a cross-chain merge. A cross-chain merge uses data from an auxiliary chain to merge two clusters that each contain at least one address of the target chain. Given that BTC is the target chain and BCH is the auxiliary chain, that corresponds to all clusters that have at least one non-green address. All other clusters only contain green addresses, which do *not* appear on the target chain, i.e., new post-fork BCH addresses. A merge of two such clusters is ineffective[4] from a cross-chain clustering perspective, as the addresses of both clusters are removed in the reduction step. In other words, a cross-chain merge is effective if it has an impact on the final, reduced target chain clustering.

4.4.2 Overview

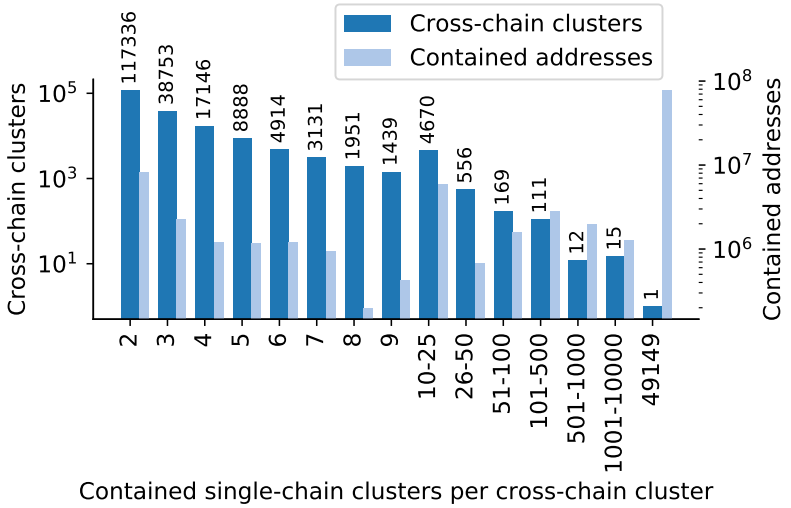
We create a cross-chain clustering of Bitcoin (target) and Bitcoin Cash (auxiliary) as of Dec 31, 2019, using the command in Listing 4.1. The output of the analysis library in Listing 4.2 already provides some interesting insights. In total, Bitcoin and Bitcoin Cash have 970 million addresses (all colors). In the reduction step 4.46% or 43 million of those addresses are excluded from the clustering because they only appear on Bitcoin Cash, i.e., green addresses. The remaining 926 million addresses appear on Bitcoin and possibly on Bitcoin Cash (all but green addresses). The resulting cross-chain Bitcoin clustering groups those 926 million addresses in 561 million clusters, i.e., every cluster contains 1.65 addresses on average.

We now focus on the benefits of adding Bitcoin Cash by comparing the created cross-chain clustering with a single-chain Bitcoin clustering. We first note that in the *single-chain* Bitcoin clustering, over 350 million clusters contain *only* post-fork BTC (yellow) addresses (approx. 465 million addresses). Cross-chain merges of these clusters by including Bitcoin Cash data are impossible. Using BCH data to enhance the BTC clustering, we observe 571,924 cross-chain merges. In other words, the cross-chain clustering has 571,924 *less* clusters than the single-chain Bitcoin clustering[5]. Every merge either links a) two single-chain clusters into a cross-chain cluster, or b) adds a single-chain cluster to an existing cross-chain cluster. In total the cross-chain clustering has almost 200,000 cross-chain clusters that contain over 770,000 single-chain clusters. Figure 4.5 quantifies the number of cross-chain clusters per quantity of contained single-chain clusters. [6]

[4] And thus, by our definition not a cross-chain merge.

[5] Each merge collapses two clusters into one and thus, reduces the number of clusters by one.

Figure 4.5: The number of cross-chain clusters and the addresses they contain, per quantity of contained single-chain clusters; as of Dec 31, 2019.



For example, 117,336 cross-chain clusters exist that are the result of two single-chain clusters being merged. Some cross-chain clusters contain hundreds or thousands of single-chain clusters. The right-most bars represent a supercluster that contains 49,149 single-chain clusters with a total of 70 million addresses. We conjecture that this cluster is the result of false positives[6]. All cross-chain clusters together contain 30 million addresses, excluding the 70 million addresses in the supercluster. These addresses are potentially affected by cross-chain clustering and account for roughly 3% of all Bitcoin addresses. [6]

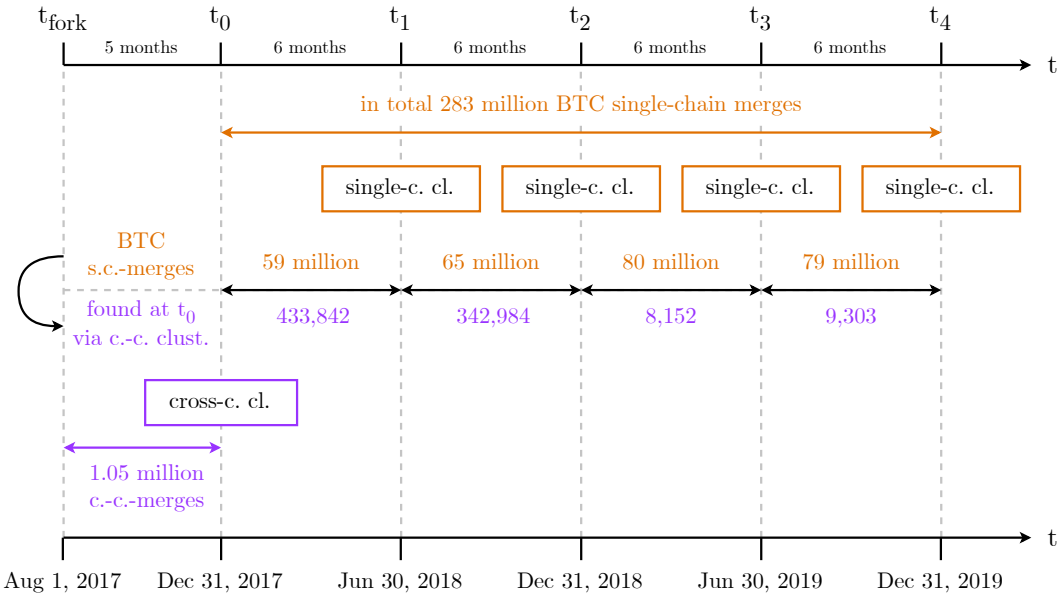
4.4.3 Time advantage

Some of the cross-chain merges that are found by including Bitcoin Cash may also occur on Bitcoin at a later point in time. We quantify the time advantage the analyst has from including Bitcoin Cash data by creating an early cross-chain clustering at t_0 (Dec 31, 2017; 5 months after the BCH fork), and comparing it to four individual BTC single-chain clusterings at $t_{1,2,3,4}$, created with snapshots every 6 months until Dec 31, 2019.[7] Figure 4.6 illustrates the analysis setup and shows first results.

[6]This assumption is backed by the fact that we could assign 34 conflicting tags, i.e., tags of unrelated entities, to this cluster using the public GraphSense address tagpack [47].

[7] t_1 = 2018-06-30, t_2 = 2018-12-31, t_3 = 2019-06-30, t_4 = 2019-12-31

Figure 4.6: Analysis setup to quantify the time advantage of including Bitcoin Cash data, as of Dec 2017.



Over 280 million merges are found in Bitcoin's single-chain clustering during t_0 and t_4. By using Bitcoin Cash as an auxiliary chain, 0.28% of those merges are found earlier by an average of 8.9 months (Table 4.1). Bitcoin Cash yields a total of 1.05 million additional cross-chain merges until t_0. The majority (75.44%) of those merges are also found in Bitcoin until Dec 2019, mostly within 12 months after t_0 – an indication that cross-chain produces a reliable time-advantage. This also implies that merges on BCH can predict future merges on BTC.

Table 4.1: Time advantage using cross-chain clustering

Period	**Merges** in BTC[1]	**Found at t_0** in BTC/BCH[2]	**Time advantage** by BCH[3] (months)
$t_0 - t_1$	59,409,904	433,842 (0.730%)	0-6m
$t_1 - t_2$	65,302,004	342,984 (0.525%)	6-12m
$t_2 - t_3$	79,627,009	8,152 (0.010%)	12-18m
$t_3 - t_4$	78,662,587	9,303 (0.012%)	18-24m
$t_0 - t_4$	282,996,504	794,281 (0.28%)	avg. 8.9m

[1]: Merges using BTC single-chain clustering.
[2]: No. of merges in [1] that were found at t_0 using cross-chain clustering.
[3]: Assuming the BCH merges were all found at t_0.

4.4.4 Privacy Implications[8]

We have shown that the activity of a user on a forked chain may cause additional cluster merges in the clustering of the parent chain. This leads to unintentional privacy compromise. We conjecture that cash-outs are a particularly common and risky behavior of users. A cash-out describes the action of transferring all owned coins to an exchange to convert them to a fiat currency. This usually involves that all owned outputs are referenced by inputs in a single cash-out transaction. Thus, all addresses of the user can be linked using the multi-input heuristic. We try to detect such cash-outs and quantify their privacy impact. In Figure 4.8 we show the address distribution between usage types over time. A small but noticeable trend is a decline in the number of addresses that existed pre-fork and initially had only been used on BCH (gray). This suggests that users may have moved their funds on the BCH chain shortly after the fork, without moving them on the BTC chain until many months after. We suspect that these may represent users who decided to cash-out their funds on the BCH chain after the fork. [6]

Figure 4.7: Address usage types on forked chains [6]

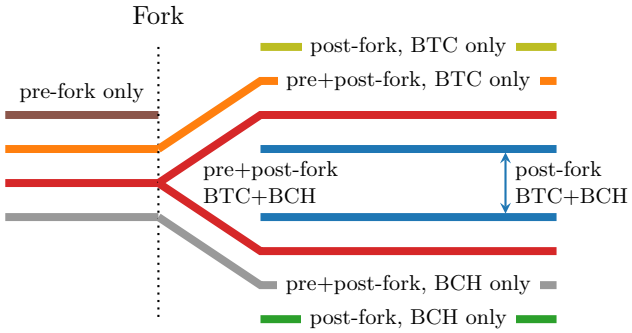
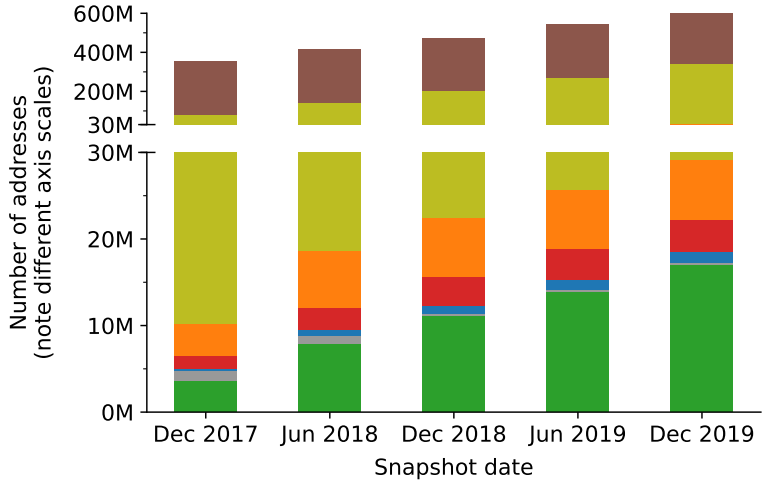


Figure 4.8: The absolute number of addresses per category. For legend and color coding see Figure 4.7. Note the difference in axis scales. [6]



[8]This section is almost verbatim from [6], esp. the 2nd paragraph and the "Address Reuse" subsection.

As in Section 4.4.3, we create an early cross-chain clustering for Dec 31, 2017, five months after the BCH fork, and compare it to individual BTC single-chain clusterings created every 6 months until Dec 31, 2019. Combining the BCH clustering with the BTC clustering yields a total of 1.05 million additional cluster merges until Dec 31, 2017. 75.44% of those early merges on BCH occur on the BTC chain (on average, about 8.9 months after occurring on the BCH chain). The high degree of overlap provides evidence that observing cluster merges on the BCH chain does indeed indicate that the corresponding BTC clusters belong to the same entity. The remaining 24.56% represent an upper bound of the unique additional privacy leakage for BTC users from their behavior on BCH as of Dec 2019. With the rough heuristic that each cross-chain cluster represents a distinct user, 99,500 users are still affected by this privacy leak over two years after the fork: that is, it becomes possible to link their BTC addresses with each other based on their BCH activity. [6, in verbatim form]

Address Reuse

Further investigating the different address use patterns, we observe the appearance of previously unseen addresses on both chains after the fork, i.e., the blue addresses in Figure 4.7. As of Dec 31, 2019, there are over one million such addresses, holding a total of 360,000 btc (USD 2.7 billion) respectively 1.45 million bch (USD 303 million). Such reuse may occur deliberately (e.g., when users import keys into wallets on both chains) or unintentionally (e.g., when hierarchical deterministic wallets continue to generate similar keys after the fork). Either way, it may not only lead to continued privacy compromise, but also raises severe security concerns. To protect their keys, those users need to enforce the same security policies on both chains, including a strict separation of keys between hot and cold wallets (e.g., avoid importing a cold wallet key into a hot wallet), as compromise of keys on one chain would allow the attacker to steal coins on all chains that share those keys (cf. [48]). [6, in verbatim form]

4.5 Discussion

We proposed and implemented a novel clustering technique that uses data of multiple chains: cross-chain address clustering. By including Bitcoin Cash data we identified over 570,000 additional cross-chain merges as of Dec 2019, i.e., merges that can not be found using single-chain clustering. We conjecture that most users are not aware that their behavior across chains can lead to privacy compromise. The effectiveness of cross-chain clustering depends on the activity of users on the forked chains. In the case of Bitcoin Cash, user interest in the fork may have been low, resulting in less usage and thus only limited utility for cross-chain clustering. We suggest that cross-chain clustering may become more useful when both forks see steady, continued use.

We also note that currently cross-chain clustering in BlockSci is limited to forked chains only. However, in a preliminary analysis we discovered that users also reuse keys across unrelated, i.e., non-forked, chains. For example, we found that 0.116% of public keys used in Litecoin have also been used on Bitcoin or Bitcoin Cash. We plan to support cross-chain clustering of non-forked chains in the future.

5 Conclusion

With the increasing frequency of blockchain forks in recent years, we have seen high potential in the analysis of those forked chains together with their parent chain. However, we have not found any publicly available tool that supports such cross-chain analyses. Existing tools allow to analyze chains individually, but do not normalize data across chains, as is needed to utilize the data relationships between forks. Thus, we decided to extend the open-source blockchain analysis platform BlockSci to meet our requirements.

We added a new multi-chain mode to BlockSci that allows user-friendly and memory-efficient cross-chain analyses of forked ledgers. The main contribution of this multi-chain mode is the deduplication of addresses across forked chains. This allows us to provide the user with addresses that are compatible between chains – a core requirement to utilize the data relationships between forked chains in cross-chain analyses. It enables the analyst to study user behavior across chains based on the activity of the same addresses on multiple chains. We evaluated the resulting extension for correctness using a custom integrity checker and found no errors. Our performance evaluation indicates that the speed is mostly on par with the non-extended version of BlockSci. Overall, the new multi-chain mode adds a robust and extensible cross-chain foundation to BlockSci. We suggest several improvements as future work. A new cross-chain API is planned that makes querying data across chains even more convenient and efficient. Further, we see potential to optimize the performance of both modes.

We used the new multi-chain mode to implement a novel address clustering technique: cross-chain address clustering. This technique uses the known heuristics and applies them to multiple related chains to create an improved clustering. Using Bitcoin Cash to improve a Bitcoin clustering we identified over 570,000 additional merges as of Dec 2019. Our analysis indicates that certain user behavior, e.g., cashing out, can compromise privacy across chains. We assume that more privacy-harming behavior across forked chains exists and suggest future work in this field. Moreover, preliminary results indicate that cross-chain clustering may even be effective across unrelated chains, e.g., when users import the same private keys in wallets of different chains. We conjecture most users are not aware that their actions across chains can negatively affect their privacy, and that importing the same private keys to multiple wallets may put their funds at risk.

Cross-chain address clustering is just one of many novel cross-chain analyses. We expect that cross-chain investigations will become more prevalent in the future. We see a trend towards multi-layer solutions in the DLP ecosystem, i.e., different blockchain systems that build on each other and interoperate. If this trend continues we expect an increasing demand for analysis techniques and tools that efficiently extract information across chains. We hope our contribution enables new types of analyses and sparks future work in this area.

List of Abbreviations

API application programming interface. 2, 31, 34–36, 47–49, 56, 58, 59, 71

BCH Bitcoin Cash. 1, 2, 34, 35, 43, 45, 48–51, 54–57, 59–61, 63–70, 73, 74

BIP Bitcoin Improvement Proposal. 12, 58

BTC Bitcoin. 1, 2, 34, 35, 40, 43, 45, 48–51, 56, 57, 59–61, 63–70, 73, 74

DLP Distributed Ledger Protocol. 4, 5, 7, 8, 11, 12, 71

DLPs Distributed Ledger Protocols. 4–9, 11, 14, 19

ETH Ethereum. 8

ETL extract, transform, load. 22

FR functional requirements. 57, 58

JSON JavaScript Object Notation. 23, 29, 30

NFR non-functional requirements. 57, 58

P2PK Pay-to-Public-Key. 9, 16, 17, 27, 40, 42, 72, 74

P2PK(H) See P2PK and P2PKH. 24, 28, 31, 42

P2PKH Pay-to-Public-Key-Hash. 8, 16, 17, 27, 42, 51, 61, 72, 74

P2SH Pay-to-Script-Hash. 12, 16, 17, 24, 27, 28, 31, 42, 58, 59, 74

P2WPKH Pay-to-Witness-Public-Key-Hash. 58, 59

RFC Request for Comments. 35

RPC Remote Procedure Call. 29, 30

SegWit Segregated Witness. 24, 58

UTXO Unspent Transaction Output. 4, 8–10, 13, 14, 23, 34, 40, 46, 73

UTXOs Unspent Transaction Outputs. 9, 13, 14, 30, 34, 38, 46, 58, 73

List of Figures

List of Listings

List of Tables

References

[1] Arvind Narayanan, Joseph Bonneau, Edward Felten, Andrew Miller, and Steven Goldfeder. *Bitcoin and Cryptocurrency Technologies: A Comprehensive Introduction*. Princeton University Press, 2016.

[2] Malte Möser and Rainer Böhme. The price of anonymity: Empirical evidence from a market for Bitcoin anonymization. *Journal of Cybersecurity*, 3(2):127–135, 2017.

[3] Malte Möser and Rainer Böhme. Trends, tips, tolls: A longitudinal study of Bitcoin transaction fees. In *International Conference on Financial Cryptography and Data Security*, pages 19–33. Springer, 2015.

[4] Harry Kalodner, Steven Goldfeder, Alishah Chator, Malte Möser, and Arvind Narayanan. BlockSci: Design and applications of a blockchain analysis platform. *arXiv preprint arXiv:1709.02489*, 2017.

[5] 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 - IMC '13*, pages 127–140, Barcelona, Spain, 2013. ACM Press.

[6] Harry Kalodner, Malte Möser, Kevin Lee, Steven Goldfeder, Martin Plattner, Alishah Chator, and Arvind Narayanan. BlockSci: Design and applications of a blockchain analysis platform. In *29th USENIX Security Symposium 2020*. ("in minor revisions" as of Jun 12, 2020), 2020.

[7] Fabian Schär. Blockchain Forks: A Formal Classification Framework and Persistency Analysis. *Working paper*, February 2020.

[8] Andreas M Antonopoulos. *Mastering Bitcoin: Unlocking Digital Cryptocurrencies*. O'Reilly Media, Inc., 2014.

[9] Yujin Kwon, Hyoungshick Kim, Jinwoo Shin, and Yongdae Kim. Bitcoin vs. Bitcoin Cash: Coexistence or Downfall of Bitcoin Cash? In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 935–951, May 2019.

[10] GitHub — Bitcoin Cash Specification: UAHF Technical Specification. `https://github.com/bitcoincashorg/bitcoincash.org/blob/master/spec/uahf-technical-spec.md`, July 2017. (accessed 2020-06-06).

[11] Satoshi Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System. October 2008.

[12] CoinMarketCap — Cryptocurrency Market Capitalizations. `https://coinmarketcap.com/`. (accessed 2020-05-08).

[13] Bitcoin Wiki — Bitcoin Script. `https://en.bitcoin.it/wiki/Script`. (accessed 2020-05-03).

[14] Forkdrop.io — Bitcoin Forks, Airdrops and Exchange Directory. `https://forkdrop.io/`. (accessed 2019-05-06).

[15] Malte Möser, Kyle Soska, Ethan Heilman, Kevin Lee, Henry Heffan, Shashvat Srivastava, Kyle Hogan, Jason Hennessey, Andrew Miller, Arvind Narayanan, and Nicolas Christin. An Empirical Analysis of Traceability in the Monero Blockchain. *Proceedings on Privacy Enhancing Technologies*, 2018(3):143–163, June 2018.

[16] George Kappos, Haaroon Yousaf, Mary Maller, and Sarah Meiklejohn. An Empirical Analysis of Anonymity in Zcash. In *27th USENIX Security Symposium 2018*, pages 463–477, 2018.

[17] Marie Vasek and Tyler Moore. There's No Free Lunch, Even Using Bitcoin: Tracking the Popularity and Profits of Virtual Currency Scams. In Rainer Böhme and Tatsuaki Okamoto, editors, *Financial Cryptography and Data Security*, Lecture Notes in Computer Science, pages 44–61, Berlin, Heidelberg, 2015. Springer.

[18] Danny Yuxing Huang, Maxwell Matthaios Aliapoulios, Vector Guo Li, Luca Invernizzi, Elie Bursztein, Kylie McRoberts, Jonathan Levin, Kirill Levchenko, Alex C. Snoeren, and Damon McCoy. Tracking Ransomware End-to-end. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 618–631, May 2018.

[19] Massimo Bartoletti and Livio Pompianu. An Analysis of Bitcoin OP_RETURN Metadata. In Michael Brenner, Kurt Rohloff, Joseph Bonneau, Andrew Miller, Peter Y.A. Ryan, Vanessa Teague, Andrea Bracciali, Massimiliano Sala, Federico Pintore, and Markus Jakobsson, editors, *Financial Cryptography and Data Security*, Lecture Notes in Computer Science, pages 218–230, Cham, 2017. Springer International Publishing.

[20] Elli Androulaki, Ghassan O Karame, Marc Roeschlin, Tobias Scherer, and Srdjan Capkun. Evaluating user privacy in bitcoin. In *International Conference on Financial Cryptography and Data Security*, pages 34–51. Springer, 2013.

[21] Maurice Herlihy. Atomic cross-chain swaps. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing*, pages 245–254. ACM, 2018.

[22] Michael Borkowski, Christoph Ritzer, Daniel McDonald, and Stefan Schulte. Caught in Chains: Claim-First Transactions for Cross-Blockchain Asset Transfers. *TU Wien: Technische Universität Wien, Tech. Rep*, 2018.

[23] Martin Harrigan, Lei Shi, and Jacob Illum. Airdrops and Privacy: A Case Study in Cross-Blockchain Analysis. In *2018 IEEE International Conference on Data Mining Workshops (ICDMW)*, pages 63–70. IEEE, 2018.

[24] Abraham Hinteregger and Bernhard Haslhofer. An Empirical Analysis of Monero Cross-Chain Traceability. In *International Conference on Financial Cryptography and Data Security*, pages 150–157. Springer, 2019.

[25] Jeremy Rubin. GitHub — BTCSpark. `https://github.com/JeremyRubin/BTCSpark`. (accessed 2020-05-10).

[26] znort987. GitHub — blockparser. `https://github.com/znort987/blockparser`. (accessed 2020-05-10).

[27] Michele Spagnuolo, Federico Maggi, and Stefano Zanero. Bitiodine: Extracting intelligence from the bitcoin network. In *International Conference on Financial Cryptography and Data Security*, pages 457–468. Springer, 2014.

[28] Michael Egger. GitHub — rusty-blockparser. `https://github.com/gcarq/rusty-blockparser`. (accessed 2020-05-10).

[29] Bernhard Haslhofer, Roman Karl, and Erwin Filtz. O Bitcoin Where Art Thou? Insight into Large-Scale Transaction Graphs. In *SEMANTiCS (Posters, Demos, SuCCESS)*, 2016.

[30] Neo4j Graph Platform. `https://neo4j.com/`. (accessed 2020-05-10).

[31] Stéphane Traumat. GitHub — blockchain2graph. `https://github.com/straumat/blockchain2graph`. (accessed 2020-06-03).

[32] Chainalysis — The Blockchain Analysis Company. `https://www.chainalysis.com/`. (accessed 2020-05-10).

[33] Elliptic Enterprises Limited — Preventing and detecting criminal activity in cryptocurrencies. `https://www.elliptic.co`. (accessed 2020-05-10).

[34] CipherTrace, Inc. — The Blockchain Security Company. `https://ciphertrace.com/`. (accessed 2020-05-10).

[35] Chainalysis — Graphing Beyond Bitcoin: Tracking the Flow of Funds for Multiple Cryptocurrencies. `https://go.chainalysis.com/multicoin.html`. (accessed 2020-05-10).

[36] Bitcoin Wiki — CoinJoin. `https://en.bitcoin.it/wiki/CoinJoin`. (accessed 2020-05-15).

[37] BlockSci Version 0.6 Documentation (without multi-chain mode). `https://mplattner.github.io/BlockSci/thesis/0.6-noext/`. (accessed 2020-06-06).

[38] Martin Plattner. GitHub — mplattner/BlockSci. `https://github.com/mplattner/BlockSci`. (accessed 2020-05-17).

[39] Facebook Open Source. RocksDB: A persistent key-value store. `http://rocksdb.org/`. (accessed 2020-05-11).

[40] Michael Kerrisk. *The Linux Programming Interface: A Linux and UNIX System Programming Handbook*. No Starch Press, San Francisco, 2010.

[41] Wenzel Jakob. GitHub — dset. `https://github.com/wjakob/dset`. (accessed 2020-05-13).

[42] Scott Bradner. Key words for use in RFCs to Indicate Requirement Levels. `https://tools.ietf.org/html/rfc2119`. (accessed 2020-05-22).

[43] Google. GitHub — sparsehash. `https://github.com/sparsehash/sparsehash`. (accessed 2020-04-16).

[44] Luke Dashjr. GitHub — BIP115: Generic anti-replay protection using Script. `https://github.com/bitcoin/bips/blob/master/bip-0115.mediawiki`, September 2016. (accessed 2020-04-23).

[45] BlockSci Version 0.6 Documentation (with multi-chain mode). `https://mplattner.github.io/BlockSci/thesis/0.6-ext/`. (accessed 2020-06-06).

[46] Eric Lombrozo, Johnson Lau, and Peter Wuille. GitHub — BIP141: Segregated Witness (Consensus layer). `https://github.com/bitcoin/bips/blob/master/bip-0141.mediawiki`, December 2015. (accessed 2020-06-06).

[47] Austrian Institute Of Technology. GitHub — graphsense-tagpacks. `https://github.com/graphsense/graphsense-tagpacks`. (accessed 2020-06-03).

[48] CCN.com — $3.2 Million Theft: Bitcoin Gold Wallet Scam Sees Fraudsters Steal Users' Private Keys. `https://www.ccn.com/bitcoin-gold-wallet-scam-nets-fraudsters-3-2-million-after-stealing-users-private-keys/`, November 2017. (accessed 2020-05-26).