BlockSci: Design and applications of a blockchain analysis platform

Harry Kalodner, Malte Möser, and Kevin Lee, *Princeton University;*
Steven Goldfeder, *Cornell Tech;* Martin Plattner, *University of Innsbruck;*
Alishah Chator, *Johns Hopkins University;* Arvind Narayanan, *Princeton University*

https://www.usenix.org/conference/usenixsecurity20/presentation/kalodner

This paper is included in the Proceedings of the
29th USENIX Security Symposium.

August 12–14, 2020

978-1-939133-17-5

Open access to the Proceedings of the
29th USENIX Security Symposium
is sponsored by USENIX.

BlockSci: Design and applications of a blockchain analysis platform

Harry Kalodner*
Princeton University

Malte Möser*
Princeton University

Kevin Lee
Princeton University

Steven Goldfeder
Cornell Tech

Martin Plattner
University of Innsbruck

Alishah Chator
Johns Hopkins University

Arvind Narayanan
Princeton University

Abstract

Analysis of blockchain data is useful for both scientific research and commercial applications. We present BlockSci, an open-source software platform for blockchain analysis. BlockSci is versatile in its support for different blockchains and analysis tasks. It incorporates an in-memory, analytical (rather than transactional) database, making it orders of magnitudes faster than using general-purpose graph databases. We describe BlockSci's design and present four analyses that illustrate its capabilities, shedding light on the security, privacy, and economics of cryptocurrencies.

1 Introduction

Public blockchains constitute an unprecedented research corpus of financial transactions. Bitcoin's blockchain alone is 260 GB as of December 2019.[1] This data holds the key to measuring the privacy of cryptocurrencies in practice, studying user behavior with regards to security and economics, or understanding the non-currency applications that use the blockchain as a database.

We present BlockSci, a software platform that enables the science of blockchains. It addresses three pain points of existing tools: poor performance, limited capabilities, and a cumbersome programming interface. Compared to the use of general-purpose graph databases, BlockSci is hundreds of times faster for sequential queries and substantially faster for all queries, including graph traversal queries. It comes bundled with analytic modules such as address clustering, exposes different blockchains through a common interface, collects "mempool" state and imports exchange rate data, and gives the programmer a choice of interfaces: a Jupyter notebook for intuitive exploration and C++ for performance-critical tasks. In contrast to commercial tools, BlockSci is not tailored to specific use cases such as criminal investigations or insights for cryptocurrency traders. Instead, by providing efficient and convenient programmatic access to the full blockchain data, it enables a wide range of reproducible, scientific analyses.

BlockSci's design starts with the observation that blockchains are append-only databases; further, the snapshots used for research are static. Thus, the ACID properties of transactional databases are unnecessary. This makes an in-memory analytical database the natural choice. On top of the obvious speed gains of memory, we apply a number of tricks such as converting hash pointers to actual pointers and deduplicating address data, which further greatly increase speed and decrease the size of the data. We plan to scale vertically as blockchains grow, and we expect that this will be straightforward for the foreseeable future, as commodity cloud instances currently offer up to a *hundred times* more memory than required for loading and analyzing Bitcoin's blockchain. Avoiding distributed processing is further motivated by the fact that blockchain data is graph-structured, and thus hard to partition effectively. In fact, we conjecture that the use of a traditional, distributed transactional database for blockchain analysis has infinite COST (Configuration that Outperforms a Single Thread) [1], in the sense that no level of parallelism can outperform an optimized single-threaded implementation.

BlockSci comes with batteries included. First, it is not limited to Bitcoin: a parsing step converts a variety of blockchains into a common, compact format. Currently supported blockchains include Bitcoin, Bitcoin Cash, Bitcoin SV, Litecoin, and Zcash (Section 2.1). A multi-chain mode optimizes for user-friendly and memory-efficient analyses of forked blockchains together with their parent chain. Smart contract platforms such as Ethereum are outside our scope.

Second, BlockSci includes a library of useful analytic tools, such as identifying special transactions (e.g., CoinJoin) and linking addresses to each other based on well-known heuristics, including across forked chains (Section 2.4). Third, BlockSci can record the time of transaction broadcasts on the peer-to-peer network and expose them through the same interface. Similarly, we make (historical and current) data on

*These authors contributed equally to this work.

[1] All numbers in this paper are current as of December 2019, and analyses of the Bitcoin blockchain as of block height 610,695, unless stated otherwise.

the exchange rates between cryptocurrencies and fiat currencies readily available. These allow many types of analyses that wouldn't be possible with blockchain data alone.

The analyst begins exploring the blockchain through a Jupyter notebook interface (Section 2.5), which initially exposes a `chain` object, representing the entire blockchain. Startup is instantaneous because transaction objects are not initially instantiated, but only when accessed. Iterating over blocks and transactions is straightforward, as illustrated by the following query, which computes the average fee paid by transactions in each block mined in December 2019:

```
fees = [mean(tx.fee for tx in block) for
        block in chain.range('Dec 2019')]
```

This interface is suitable for exploration, but for analyses requiring high performance, BlockSci also has a C++ interface. For many tasks, most of the code can be written in Python using a "fluent interface", an API design pattern that combines expressiveness and high performance (Section 2.5).

In Section 3 we present four applications to illustrate the capabilities of BlockSci. First, we show how multisignatures have the unfortunate effect of weakening confidentiality by exposing the details of access control on the blockchain, and hurting the privacy of users who *do not* use them (Section 3.1). Next, we study how users' cash-out behavior after the Bitcoin Cash hard fork hurt their privacy (Section 3.2) and find patterns of key reuse that may put users' funds at elevated risk. Turning to economics, we analyze Bitcoin Core's fee estimation's algorithm (Section 3.3), and find it relatively ineffective for predicting waiting times due to the inherent uncertainty of proof-of-work. Finally, we provide improved estimates of the velocity of cryptocurrencies, i.e., the frequency with which coins change possession (Section 3.4). This helps us understand their use as a store of value versus a medium of exchange.

2 Design and architecture

Figure 1 shows an overview of BlockSci's architecture. There are two routes for importing data into BlockSci (Section 2.1). Through either route, the data is converted by the parser (Section 2.2) into the BlockSci Data (Section 2.3), which can be incrementally updated as new blocks come in. The analysis library (Section 2.4) loads this data as an in-memory database, which the user can either query directly (in C++) or through a Jupyter notebook interface (Section 2.5).

A recurring theme in this section is that since BlockSci is a domain-specific database, we are able to make assumptions about the schema and the workload that allow us to achieve large performance gains and an expressive interface. Both this broad lesson and some of our specific optimizations may be applicable to other domains.

2.1 Recording and importing data

Design decision: which blockchains should BlockSci support? There are hundreds of blockchains, some of which differ from Bitcoin in minor ways and others drastically. As we aim to provide a common interface for the analysis of all supported blockchains, supporting too few blockchains (e.g., just Bitcoin) limits usefulness, but supporting too many different blockchains would complicate the interface and make optimizations harder.

Recall that the Bitcoin blockchain consists primarily of a directed acyclic graph of transactions. The edges connecting transactions have attributes, i.e., addresses or scripts, attached to them. Transactions are grouped into blocks which are arranged in a linear chain, with a small amount of metadata per block. BlockSci supports blockchains that follow this basic structure. For example, Litecoin makes no changes to the data structure, and is thus fully supported. Cryptocurrencies that introduce changes to the script operations may be supported only partially, but the user can parse unknown scripts with a few lines of code. Zcash is also supported, at least to the extent that Zcash blockchain analysis is even possible: it introduces a complex script that includes zero-knowledge proofs, but these aspects are parceled away in a special type of address that is not publicly legible by design.

An example of an unsupported blockchain is Monero, as it doesn't follow the "each input spends one output" paradigm. Its transaction graph contains additional edges, the mixins. Supporting it would require changes to the data layout as well as the programmer interface. Similarly, Ethereum departs from the transaction-graph model, and further, its script is vastly different from and more complex than that of Bitcoin.

In our analyses we have worked with six blockchains: Bitcoin, Bitcoin Cash, Litecoin, Namecoin, Dash, and Zcash. Many other cryptocurrencies make no changes to the blockchain format, and so should be supported with no changes to BlockSci.

Multi-chain mode. By default, BlockSci operates on a single blockchain. We also provide a multi-chain mode in which several forked chains (e.g., Bitcoin $\prec$ Bitcoin Cash $\prec$ Bitcoin SV) can be combined in an optimized, memory-efficient multi-chain configuration. In this mode, data common to forked chains (such as pre-fork transactions) need to be loaded into memory only once. Address data is deduplicated across forks, allowing for novel cross-chain analyses.

Importer. For cryptocurrencies with small blockchains where import performance is not a concern, we use the JSON-RPC interface. The advantage of this approach is versatility, as many cryptocurrencies aim to conform to a standard JSON-RPC schema regardless of the on-disk data structures and serialization format. For larger blockchains (currently only Bitcoin and its forks are large enough for import performance to be a concern), we use our own high-performance importer that directly reads the raw data on disk.

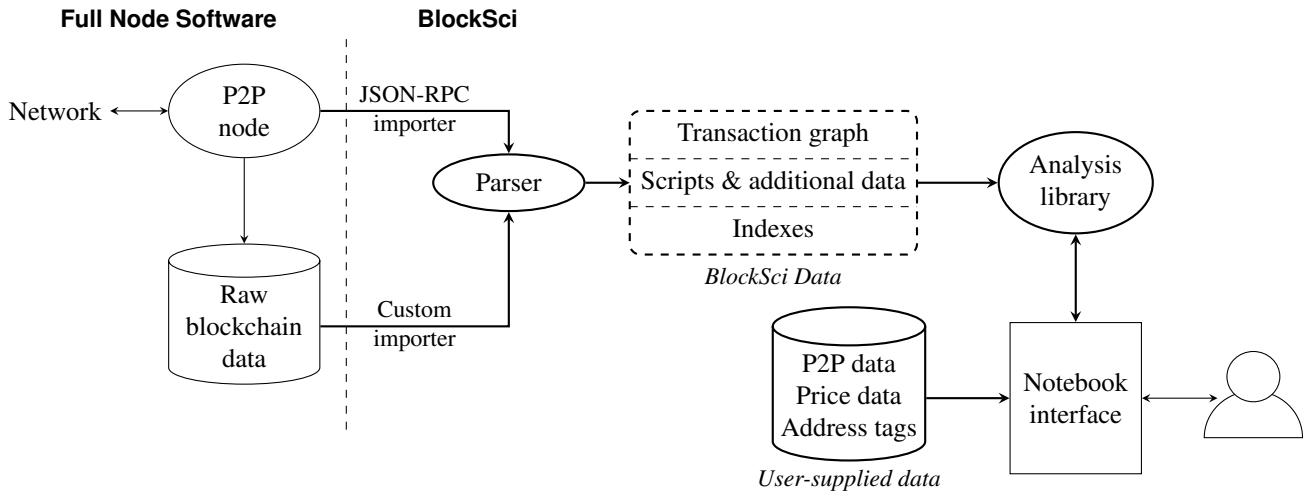

Figure 1: Overview of BlockSci's architecture.

Mempool recorder. BlockSci can optionally record mempool data, that is, timestamps of transactions that are broadcast to the P2P network and are waiting to be included in the blockchain. The waiting time of included transactions provides valuable data for economic analyses and isn't recorded in the blockchain itself. When users choose to collect these timestamps, they are accessible through the same interface as all other blockchain data.

2.2 Parser

Implementation challenge: optimizing the parser. The on-disk format of blockchains is not very usable for analysis. It is optimized for a different set of goals, such as transaction validation and data retrieval in a distributed network. Bitcoin Core and similar clients minimize memory consumption and store blocks in raw network format on disk, whereas we aim for a representation of the data that can fit in memory. Given that a data transformation is necessary, we describe the design and optimization of the parser that handles this step.

Parsing is sequential and stateful. The blockchain must be processed sequentially as two types of state are required for the transformation: one is to link a transaction's inputs to outputs of prior transactions, and the other is to link inputs/outputs to addresses. Each transaction input specifies which output it spends, encoded as (transaction hash, output index). The parser assigns an ID to every transaction and stores information for every unspent output (UTXO), including the hash $\rightarrow$ ID mapping. Similarly, it must assign IDs to addresses and maintain this mapping for linking and deduplication.

In Bitcoin, transactions spending outputs of other transactions in the same block must appear after them. Other cryptocurrencies, however, violate this rule. Bitcoin Cash uses canonical transaction ordering (CTOR) within each block, i.e., based on their hash. Thus, to process a block, the parser processes transactions in multiple passes: it first identifies all transactions in a block before it can correctly link transaction inputs to the outputs spent. This allows the parser to tolerate an arbitrary ordering of transactions within each block.

UTXOs can be removed from the parser state after they have been spent. Address mapping, however, allows no such optimization. Any address may be used by any output and thus all addresses must be tracked at all times. Storing the map in memory would require too much memory, and storing it on disk would make the parser too slow.

Optimization: Bloom filters and address caches. To achieve further optimizations, we observe that the vast majority of inputs spend recently created outputs (e.g., 88 % of inputs spend outputs created in the last 4000 blocks). And only 8.6 % of Bitcoin addresses are used more than once, but those account for 51 % of all occurrences. This motivates the following trade-off between speed and memory consumption:

1. A bloom filter (a probabilistic data structure that allows testing membership in a set) stores all seen addresses. Recall that negative results from a bloom filter are always correct, whereas there is a small chance of false positives. This ensures correctness of lookups for existing addresses while minimizing the number of database queries for nonexistent ones.
2. A multi-use address cache contains (and does not evict) all addresses that have been used multiple times.
3. Address hashes are stored in a key-value database on disk (RocksDB [2]), with a default cache that has a Least Recently Used (LRU) replacement policy. New entries are cached before being written to the database in batches.

Shared state across chains. In multi-chain mode, the parser processes all—parent and forked—chains sequentially. It shares and reuses parser states across chains, such as the bloom filter of seen addresses. By sharing a single database, address data is deduplicated across forked chains.

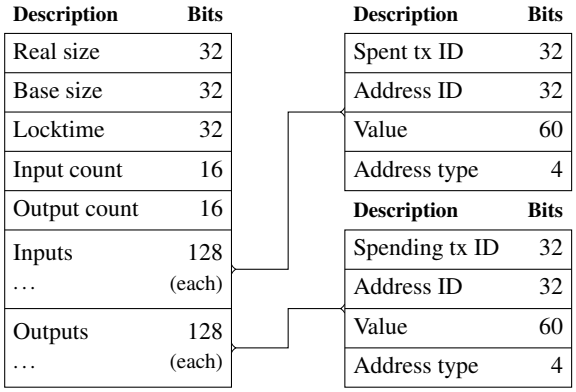

Figure 2: Transaction structure

Incremental updates. The append-only nature of the blockchain enables incremental updates to the parser output. The parser serializes its final state at the end of a run and resumes from that state when invoked again. A difficulty with this approach is handling blockchain reorganization which occurs when a block that was originally in the longest branch is surpassed by a different branch. BlockSci recommends to ignore the most recent few blocks during initialization. The probability of a reorg that affects d or more blocks decreases exponentially in d. The default value of d for Bitcoin is 6. If a deeper reorg happens, the user needs to reparse the chain.

2.3 BlockSci Data

Key challenge: finding a data layout that gives a good trade-off between memory efficiency and performance.

Based on our experience with empirical blockchain analysis over several years, we divide the available data into three categories and combine it in a hybrid scheme that provides us with a reasonable trade-off between efficient use of memory and speed of access:

1. The core transaction graph is required for most analyses and always loaded in-memory. It is stored in a row-based format.
2. Scripts and additional data is required for only a subset of analyses. It is stored in a hybrid (partially column-based, partially row-based) format and is loaded on-demand.
3. Indexes to look up individual transactions or addresses by hash are stored in a separate database on disk.

We make further optimizations to improve performance, including using fixed-size encodings for data fields where possible, optimizing the memory layout for locality of reference, linking outputs to inputs for efficient traversal, and sharing identical data across chains in multi-chain mode.

Transaction graph. The core transaction graph is stored in a single sequential table of transactions, with entries having the structure shown in Figure 2. Note that entries have variable lengths, due to the variable number of inputs and outputs (there is a separate array of offsets for indexing, due to the variable entry lengths). Normally this would necessitate entries to be allocated in the heap, rather than contiguously, which would have worse memory consumption and worse locality of reference.

However, because of the append-only property of the blockchain, there are only two types of modifications that are made to the transactions table: appending entries (due to new transactions) and length-preserving edits to existing entries (when existing UTXOs are consumed by new transactions). This allows us to create a table that is stored as flat file on disk that grows linearly as new blocks are created. To load the file for analysis, it is mapped into memory. The on-disk representation continues to grow (and be modified in place), but the analysis library provides a static view (Section 2.4).

Layout and locality. The main advantage of the transaction graph layout is spatial locality of reference. Analyses that iterate over transactions block-by-block exhibit strong locality and benefit from caching. Such analyses will remain feasible even on machines with insufficient memory to load the entire transaction graph, because disk access will be sequential.

The layout stores both inputs and outputs as part of a transaction, resulting in a small amount of duplication (a space cost of about 19 %), but resulting in a significant speedup for sequential iteration compared to a normalized layout. Variants of the layout are possible depending on the types of iteration for which we wish to optimize performance (Section 2.6).

Additional data. Beyond the core transaction graph, BlockSci provides access to additional data that are necessary for some types of analyses. These include script data, transaction hashes and version numbers, input sequence numbers, input-output linkages, and raw data contained in coinbase transactions. Keeping this data separate reduces memory usage in exchange for a small reduction in speed of access for analyses that require this data (e.g., 10 % slower for a typical query that iterates over transaction metadata).

Scripts. BlockSci categorizes scripts into 5 generic types, each of which contains scripts of one or more address formats: script-hash (for script-hash and witness-script-hash scripts), pubkey (for raw pubkey, pubkey-hash, individual pubkeys in a multisig script, and witness-pubkey-hash scripts), multisig, null data, and unknown witness scripts. All other scripts are categorized as nonstandard. Internally, script data of different address formats is deduplicated: for example, a public key used in both a pubkey-hash and a witness-pubkey-hash script is stored only once. For nonstandard scripts, BlockSci stores the entire script data which can be parsed with only a few lines of code by the analyst.

Indexes. Transaction hashes and addresses are stored in flat files and can easily be looked up by transaction/address ID. The reverse mapping from hash to ID, however, is stored in separate indexes in RocksDB databases (the address index is also used by the parser). Accessing these indexes is almost never performance critical in scientific analysis—in

fact, many analyses don't require the indexes at all. Besides the ability to look up transactions and addresses by hash, we also provide a lookup for all outputs associated with specific addresses.

Multi-chain mode. To support forked blockchains, we make three modifications to the layout described above. First, forked chains often share a large common history with their parent chain. We load these identical blocks only once, and the analysis library provides the abstraction of a full chain for each fork. Second, the fixed-size encoding does not permit storing data of multiple chains. For example, UTXOs at fork height can be spent in both the parent and the forked chains, but the fixed-length field can only hold a single index for the spending transaction (cf. Figure 2). Each fork thus needs a separate flat file that contains the spending transactions' IDs for outputs created before the fork. Third, the index that maps addresses to outputs requires an additional chain identifier to distinguish between outputs on different chains.

2.4 BlockSci Analysis Library

The snapshot illusion. The following three seemingly contradictory properties hold in BlockSci:

1. The transactions table is regularly updated on disk as new blocks are received (note that arbitrarily old transactions may be updated if they have unspent outputs that get spent in new blocks).
2. The table is memory-mapped and shared between all running instances of BlockSci.
3. Each instance loads a snapshot of the blockchain that never changes unless the programmer explicitly invokes a reload.

The contradiction disappears once we notice that the state of the transactions table at any past point in time (block height) can be reconstructed given the current state. To provide the illusion of a static data structure, when the blockchain object is initialized, it stores the height of the blockchain at initialization time. The blockchain on disk increases over time, but the stored height remains fixed, and accesses to blocks past this height are prevented. The analysis library intercepts and rewrites accesses to transaction outputs such that outputs that were spent in blocks added after initialization appear unspent.

Memory mapping and parallelism. Since BlockSci uses the same format for the transaction graph on disk and in memory, loading the blockchain simply involves memory-mapping this file. Once in memory, each transaction can be accessed as a C++ `struct`; no new memory needs to be allocated to enable an objected-oriented interface to the data. This is because the disk layout of each struct is identical to its memory layout.

Memory mapping allows users to efficiently run BlockSci on machines with less than the recommended amount of memory provided that they only require access to a subset of the data that fits in memory.

Memory mapping also allows multithreaded parallel processing with no additional effort. Recall that if a file is mapped into memory by multiple processes, they use the same physical memory for the file. The file has only one writer (the parser); it is not modified by the analysis library. Thus, synchronization between different analysis instances isn't necessary. With a disk-based database, analyses tend to be I/O-bound, with little or no benefit from multiple CPUs, whereas BlockSci is CPU-bound, and performance scales roughly linearly with the number of virtual CPUs (Section 2.6). Finally, memory mapping also makes it straightforward to support multiple users on a single machine, which is especially useful given that Jupyter notebook (the main interface to BlockSci) can be exposed via the web.

Mapreduce. Many analyses, such as computing the average transaction fee over time, can be expressed as mapreduce operations over the transactions table (or ranges of blocks). Thus the analysis library supports a mapreduce abstraction that, with no additional effort from the programmer, handles parallelizing the task to utilize all available cores. As we show in Section 2.6.1, parallel iteration over all transactions, transaction inputs, and transaction outputs on the Bitcoin blockchain as of December 2019 takes only 0.9 seconds on a single 16-vCPU EC2 instance.

Address linking. Address linking (or clustering) is a key step in many analytic tasks including understanding trends over time and evaluating privacy. Recall that cryptocurrency users can trivially generate new addresses, and most wallets take advantage of this ability. Nevertheless, addresses controlled by the same user or entity may be linked to each other, albeit imperfectly, through various heuristics.

Two common types of heuristics include (1) inputs spent in the same transaction are controlled by the same entity, and (2) identifying a change address based on client software or user behavior (e.g., [3]). As the multi-input heuristic does not apply to CoinJoin transactions, we add an exception for those transactions, which we identify using the algorithm described by Goldfeder at al. [4]. Change address identification is challenging due to the variety of existing client software. BlockSci comes with several—as of this writing, ten—change address heuristics that can be used individually or in combination with each other, allowing the analyst to choose or create a heuristic best suited for their analysis task.

These heuristics create links (edges) in a graph of addresses. By iterating over all transactions and applying the union-find algorithm on the contained addresses we generate clusters of addresses. This set of clusters is the output of address linking. We use the union-find implementation by Jakob [5]. Clustering takes only a few minutes, allowing the analyst to recompute and compare clusters with different heuristics.

In multi-chain mode, BlockSci can enhance the clustering of a target chain using information from forked chains. Addresses that exist on multiple chains may be used differently on them, e.g., combined with a different set of input addresses.

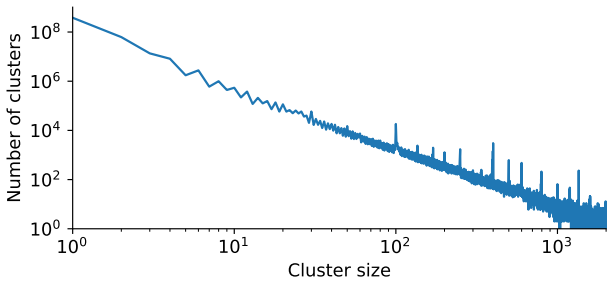

Figure 3: Distribution of sizes of address clusters in Bitcoin after applying address-linking heuristics. Sizes 1–2,000 are shown here but there are many clusters that are much larger.

Cross-chain address clustering uses these additional links to enhance the clustering of the target chain (cf. Section 3.2).

Figure 3 shows the distribution of cluster sizes for Bitcoin using the multi-input heuristic only. There are about 474 million clusters in total, of which about 380 million are single addresses, and about 93 million have between 2 and 20,000 addresses. There are 809 clusters with over 20,000 addresses, including one supercluster with over 17 million addresses.

Address linking is inherently imperfect, and ground truth is difficult to obtain on a large scale, since it requires interacting with service providers. We do not attempt to be comprehensive, resulting in false negatives (i.e., missed edges, resulting in more clusters than truly exist). More perniciously, most of the heuristics are also subject to false positives (i.e., spurious edges), which can lead to "cluster collapse". In particular, it is likely that the supercluster above is a result of such a collapse.

Tagging. Address linking is especially powerful when combined with address tagging, i.e., labeling addresses with real-world identities. This can be useful for forensics and law-enforcement investigations but it can also violate user privacy. BlockSci does not provide address tags. Tagging requires interacting with service providers and cannot be done in an automated way on a large scale. Companies such as Chainalysis and Elliptic specialize in tagging and forensics, blockchain.info allows users to publicly tag addresses that they control, and researchers sometimes provide datasets of address tags [6]. BlockSci has a limited tagging feature: if the user provides tags for a subset of addresses, individual clusters can return tags associated with them.

2.5 Programmer interface

Key challenge: combining speed and expressiveness. BlockSci aims to come close to the speed of C++ while providing expressiveness and convenience of a high-level language, namely Python, for as many analysis tasks as possible.

Python interface. Jupyter notebook is a popular Python interface for data science. It allows packaging together code, visualization, and documentation, enabling easy sharing and reproducibility of scientific findings. We expose the C++ BlockSci library to Python through the pybind11 interface [7]. While we intend Jupyter notebook to be the main interface to BlockSci, it is straightforward to utilize the analysis library directly from standalone C++ or Python programs and derive most of the benefits of BlockSci.

Python is not a language known for performance; unsurprisingly, we find that it is significantly slower to run queries through the Python interface. Nevertheless, our goal is to allow the programmer to spend most of their time interacting with the Jupyter notebook, while simultaneously ensuring that the bottleneck parts of queries execute as C++ code. We illustrate this through an example.

Suppose our goal is to find transactions with anomalously high transaction fees — say 0.1 bitcoins (10^7 satoshis), worth about 720 US dollars as of December 2019. The slowest way to do this would be to write the entire query in Python:

```
[tx for block in chain for tx in block if
    sum(txin.value for txin in tx.inputs) -
    sum(txout.value for txout in
    tx.outputs) > 10**7]
```

This way does not result in acceptable performance. A first step to improve both performance and conciseness is to have a built-in function to compute the fee:

```
[tx for block in chain for tx in block if
    tx.fee > 10**7]
```

Although `tx.fee` calls a C++ function, we model it as a property in the Python interface. Most helper functions are modeled as properties, unless they are expected to take significant time to compute, or take arguments. `tx.fee` is just one of many helpers exposed by the Python library that execute as C++. We've found that most of the analyses in Section 3 benefit from a small number of helper functions.

Fluent interface. Running this analysis over the entire blockchain in Python still does not provide great performance. At the time of writing, the Bitcoin blockchain contains more than 480 million transactions, for each of which the above query instantiates a Python object, even though only a few thousand transactions will eventually be selected.

To make analyses faster without requiring the user to write complicated C++ code, we've developed a fluent interface [8] to specify graph queries. A fluent interface is an internal domain-specific language (DSL) that allows the analyst to specify queries as a sequence of selections and filters over the transaction graph. Method chaining makes specifying sequences of operations convenient: every operation returns a proxy object to which further operations can be applied. Execution happens lazily for most parts of this interface: either when the analyst requests a list of the results or when the query reaches a point that does not allow further traversal (e.g., after selecting the fee of a transaction). Using the flu-

Table 1: BlockSci C++ running time for various queries iterating over 610,695 Bitcoin blocks.

Iterating over	Single-threaded	Multithreaded
Tx headers	6.7 sec	0.6 sec
Tx outputs	9.8 sec	0.8 sec
Tx inputs & outputs	11.3 sec	0.9 sec
Headers in random order	179.1 sec	Unsupported

Table 2: BlockSci Python running time for the anomalous-fee query iterating over 610,695 blocks under the three paradigms discussed in Section 2.5.

Query type	Single threaded	Multithreaded
Pure Python	—	18 hrs
C++ builtin	6 min 59 sec	58.6 sec
Fluent interface	38.3 sec	8.7 sec

ent interface, the anomalous-fee query can be expressed as follows:

```
chain.blocks.txes.where(lambda tx: tx.fee >
    10**7).to_list()
```

Our interface provides many options to select and filter data. The `select` clause allows to select properties of objects, though most properties can be conveniently accessed directly, e.g., `txes.fee` instead of `txes.select(lambda tx: tx.fee)`, as the library redirects such property accesses to the corresponding `select` function call. As demonstrated above, `where` filters objects using predicates. `any` and `all` apply predicates over a list of objects and return whether they apply to any or all contained items, `max` and `min` select elements with the highest (or lowest) attribute value, and a `group_by` clause returns aggregates of properties. We provide a detailed overview of available clauses in the online documentation.

The fluent interface operates single-threaded. Providing the mapreduce functionalities of the C++ interface for the fluent interface is planned for future versions. Currently, the user can work around this limitation using Python's multiprocess library to parallelize the computation on subsets of blocks or transactions.

2.6 Performance evaluation

We now report the speed and memory consumption of BlockSci. All measurements assume that the in-memory data structures are already loaded in memory. This takes about 4 minutes for Bitcoin and needs to be done only once per boot.

2.6.1 Basic run time statistics

We run measurements on a single *r5.4xlarge* EC2 instance (16 vCPUs, 2.5 GHz Intel Xeon Platinum 8175M, 128 GiB memory, 800 GiB EBS volume). The cost is $1.12 per hour.

The most common type of access is a mapreduce-style iteration over the blockchain. A representative example is finding transactions with anomalously high fees, because computing the fee requires iterating over not just transactions, but also the inputs and outputs of each transaction. In essence, this query touches the entirety of the transactions table data.

As Table 1 shows, a *single-threaded* implementation of this query completes in 11.3 seconds. Mapreduce-style queries are embarrassingly parallel, as seen in the table. Our test machine has 16 virtual cores, i.e., 8 physical cores with hyperthreading. Executed in parallel, the query finishes under one second.

The table shows that iterating over only the outputs (e.g., finding the max output value) is faster, and iterating over only the headers is faster still. The above queries benefit from locality of reference. Other queries, especially those involving graph traversal, will not. To simulate this, we iterate over transaction headers in random order. We see that there is a 26-fold slowdown.

In Section 2.5 we presented several paradigms for querying the blockchain from the Python interface: pure Python, C++ helper functions, and the fluent interface. Table 2 shows the performance of these three paradigms on the anomalous-fee query. The pure-Python method has unacceptable performance (this is partially a result of a few performance traps in the current codebase). Using the helper method is a lot faster, but using the fluent interface is preferred: it is 7–11x faster than the helper method. Compared to implementing a single-threaded C++ query, the fluent interface is only 3-5x slower for many practical applications (cf. Table 3).

2.6.2 Comparison with graph databases

Graph traversal is integral to many blockchain analyses, such as inspecting specific addresses or determining change based on properties of the spending transaction. In this section, we compare BlockSci's performance against three general-purpose graph databases: Neo4j, RedisGraph, and Memgraph.

Neo4j is one of the most popular graph databases currently available. While it is not an in-memory database, we can load the entire dataset into the page cache before executing queries. Memgraph and RedisGraph are pure in-memory graph databases, the latter being built on top of the key-value database Redis.

All three databases allow to import data in CSV format and to execute queries using the Cypher query language. This allows us to run almost exact queries on all three databases. We created an export tool for BlockSci that exports blockchain data into the CSV format readable by these databases.

The graph representation of these databases is significantly larger than the BlockSci Data format (and thus needs more disk space and memory), even though we choose to only store a few properties and not all information available in

Table 3: Average running time in seconds (over five consecutive runs) for graph queries on a dataset with 25 million transactions (up to block height 262,176). Standard deviations and running times for more datasets are provided in Table 8 in the appendix.

Query	BlockSci			Neo4j		RedisGraph	Memgraph
	C++ (ST)	C++ (MT)	Fluent interface (ST)	w/o index	w/ index		
Tx locktime > 0	0.31	0.03	1.37	7.84	0.05	1.85	16.44
Max output value	0.46	0.03	3.91	26.63	24.55	4.48	40.08
Calculate fee	0.57	0.03	2.79	302.73	303.69	–[1]	187.02
Satoshi Dice address	0.49	N/A	0.54	0.95	0.99	2.56	45.91
Zero-conf outputs	5.47	0.32	18.17	192.01	207.41	1488.93	59.96
Locktime change	7.57	0.45	18.21	208.95	213.59	–[1]	122.98

–[1]: did not finish within reasonable time (based on other queries and dataset sizes), ST = single-threaded, MT = multi-threaded

BlockSci. We deem this a reasonable compromise: while BlockSci aims to be a general-purpose tool, analysts may decide to ignore data irrelevant to their goals when choosing a different database. We design the graph property model for flexibility and expressiveness of traversal queries, thus we explicitly model blocks, transactions, outputs, and addresses as nodes. A detailed description of the model can be found in Appendix A.

We evaluate these databases on graphs of different sizes (i.e. historic snapshots of the blockchain). While we intended to run the measurements on the full transaction graph, performance issues of the graph databases already became apparent with data set sizes significantly smaller than the full graph and prevented the completion of measurements on the full graph in a reasonable time frame (cf. Table 8 in the appendix).

We run measurements on an *r5.8xlarge* EC2 instance (32 vCPUs, 256GiB memory). Besides repeating the iterative queries from the previous section (finding transactions with a positive locktime, finding the highest output value and finding the highest transaction fee) we also run three queries involving graph traversal (calculating the total value received by a popular address, counting the number of outputs that have been spent in the same block and identifying transactions where exactly one output has been spent in a transaction that uses a similar locktime policy).[2]

Table 3 shows query running times for a dataset of 25 million transactions (the current blockchain contains more than 489 million transactions). We can see that BlockSci's is generally much faster than the other databases, by a factor of 2–16 compared to the best results for graph traversal, and hundreds times faster for many sequential queries. Results for more data sets can be found in Table 8 in the appendix.

[2]The Cypher queries used are listed in Table 7 in the appendix.

Table 4: Size of the transaction graph under each of 4 possible memory layouts. The 'Current' column refers to the Bitcoin blockchain as of the end of December 2019, which has about 489 million transactions, 1.198 billion inputs and 1.302 billion outputs (including unspent ones).

	Growth (bytes)	Current
Current	$24\,N_{tx} + 16\,N_{in} + 16\,N_{out}$	50.09 GB
Normalized	$24\,N_{tx} + 8\,N_{in} + 16\,N_{out}$	40.50 GB
Fee cached	$32\,N_{tx} + 16\,N_{in} + 16\,N_{out}$	54.00 GB
64-bit	$24\,N_{tx} + 24\,N_{in} + 24\,N_{out}$	69.26 GB

2.6.3 Comparison with other open-source blockchain analysis tools

When we initially made BlockSci publicly available, we evaluated its performance against other open-source blockchain analysis tools. We found BlockSci to be 15-600x faster than these tools [9], and its performance has improved considerably since. As we attempted to repeat the comparison, we found that these tools are no longer maintained. A few new blockchain analysis tools are available, but we found that they aren't general purpose tools but only support specific use cases.

2.6.4 Parser performance

Parsing the blockchain needs to be done only once upon installation; incremental updates are essentially instantaneous. On our *r5.4xlarge* machine, parsing the Bitcoin blockchain until end of December 2019 (block height 610,695) took 5.5 hours. Note that it takes Bitcoin Core several hours to download the blockchain, so initialization is slow anyway.

2.6.5 Memory

Table 4 shows the memory consumption of BlockSci as a function of the size of the blockchain (measured by the number of transactions, inputs, outputs, and addresses). As noted

earlier, for almost all analysis tasks we have encountered so far, only the transaction table needs to be in memory to ensure optimal performance. As of December 2019, this comes out to about 50 GB for Bitcoin.

Recall that BlockSci's default layout of the transaction table is not normalized: coins are stored once as inputs and once as outputs. The table also shows the memory consumption for several alternate layouts. While normalizing the layout would save 19 % space, it leads to a steep drop in performance for typical queries such as max-fee. Alternatively, we could store derived data about transactions, such as the fee, at the expense of space. Finally, we also show how the space consumption would increase if and when we need to transition to 64-bit integers for storing transaction and address IDs.

3 Applications

We now present four analyses that highlight BlockSci's effectiveness at supporting blockchain analyses. The first two relate to privacy and confidentiality, the third and fourth to the economics of cryptocurrencies. Table 5 shows how these applications take advantage of the features of BlockSci's analysis library and data sources.

3.1 Multisignatures hurt confidentiality

Security conscious users or companies that store large amounts of cryptocurrency often make use of Bitcoin's multisignature capability. Unlike standard pay-to-public-key-hash (P2PKH) transactions which only require one signature to sign, multisig addresses allow one to specify n keys and a parameter $m \leq n$ such that any m of the specified keys must sign in order to spend the money. This feature allows distributing control of a Bitcoin wallet: keys can be stored on n servers or by n different employees of a company such that m of them must agree to authorize a transaction. Similarly, a user could store a key on both her desktop computer and her smartphone and require the participation of both to authorize a transaction (a 2-of-2 multisig).

Bitcoin's multisig implementation requires users to explicitly list all n keys as well as the values m and n. To make it easier to receive funds to multisig addresses, Bitcoin implements an address format called pay-to-script-hash (P2SH), where the sender only needs to know a hash value of the full script. When spending from such an address, the receiver has to provide all individual keys and the parameters m and n along with valid signatures in the input. As of December 2019, up to 27 % of all bitcoins mined are held in multisig addresses.[3]

[3] There is some uncertainty because we can only determine whether a P2SH script wraps a multisig script or some other kind of script once it has been spent. However, past data suggests that most of the value in P2SH outputs indeed correspond to multisig.

In this section we show how multisignatures expose confidential information about access control on the blockchain, as suggested by Gennaro et al [13]. We further show how the use of multisignatures can hurt the privacy of *other* users. Finally, we observe patterns of multisig usage that substantially reduce its security benefits.

Confidentiality. For companies or individuals that use multisig to enforce access control over their wallet, multisig publicly exposes the access control structure as well as changes to that structure. In other words, it exposes the number of total keys and the number of keys needed to sign, the individual (public) keys themselves, as well as changes in access control that may correspond to events such as a loss of a device or the departure of an employee.

Two characteristics indicate that a transaction might represent a change in access control:

- Single input, single output. Payment transactions typically involve multiple inputs and/or change outputs. By contrast, a transaction with only one input and one output (whether a regular or a multisig address) suggests that both are controlled by the same entity.
- Overlapping sets of multisig keys between the input and the output suggest a change in access control (e.g., the replacement or removal of a specific key), but not a complete transfer of control.

As an example of such a transaction with these characteristics, consider the transaction 96d95e...[4]. In this transaction, over USD $130,000 of Bitcoin was transferred from one 2-of-3 multisig address to a second 2-of-3 multisig address. These addresses shared 2 keys in common, but one of the original keys was replaced with a different key. Chainalysis[5] labels both the input and output addresses as being controlled by `coinsbank.com`. This publicly reveals an internal access control change happening at a private company.

Figure 4 shows that these types of information leakage happen regularly. Every month, tens of thousands of transactions transferring bitcoins worth millions of dollars publicly expose confidential access control structure changes in this way.

Privacy. When an output address uses the same type of access-control policy as an input address, it is a strong indicator that the output was used as a change address. This provides a powerful heuristic to identify change addresses. We find that for many transactions, this heuristic allows identifying change addresses even though previously known heuristics (e.g., [3]) do not allow such a determination.

While Gennaro et al. mention the unfortunate privacy-infringing side-effect of multisig [13], we provide the first empirical evidence for the pervasiveness of this effect. We have implemented a generalized heuristic that identifies the change address based on it being the only output that matches

[4] https://blockchain.info/tx/96d95eb77ae1663ee6a6dbcebbbd4fc7d7e49d4784ffd9f5e1f3be6cd5f3a978

[5] https://www.chainalysis.com/

Application	Transaction graph analysis	Address linkage (clustering)	CoinJoin detection	Script parsing	Mempool data	Exchange rate data	Multi-chain mode	Altcoin support
Multisignature transactions (Section 3.1)	•	•		•		•		
Multi-chain privacy (Section 3.2)	•	•	•				•	•
Fee estimation effectiveness (Section 3.3)	•				•			
Velocity of cryptocurrencies (Section 3.4)	•	•				•		
Selected papers (cf. Section 3.5)								
Privacy and linkability of mining in Zcash [10]	•							•*
Tracking ransomware end to end [11]			•					
Tracing transactions across cryptocurrency ledgers [12]	•							•
When the cookie meets the blockchain [4]	•	•	•			•		•

* implements additional functionality for Zcash on top of BlockSci

Table 5: Usage of BlockSci features and data sources in our analyses and selected papers.

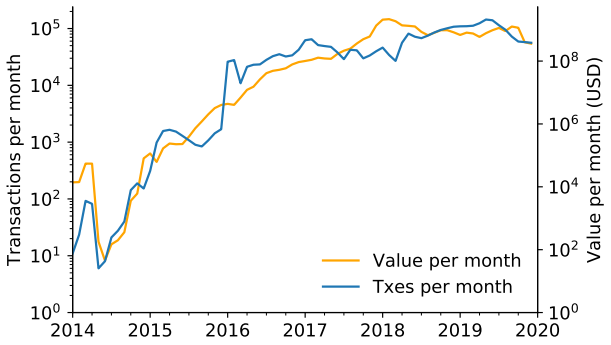

Figure 4: Frequency and value of multisig transactions that expose confidential information about access structure changes on the blockchain.

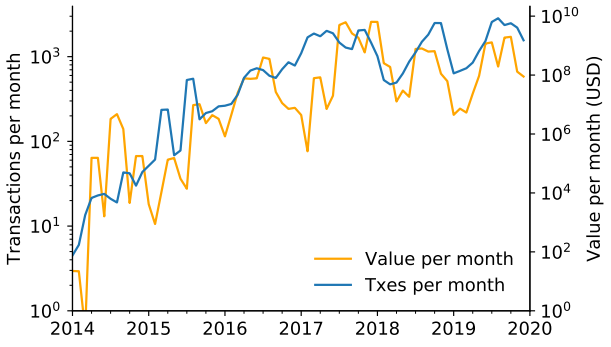

Figure 5: Frequency and value of transactions that weaken multisig security by temporarily sending coins to regular addresses, advertising the presence of a single point of failure.

the type of all input addresses (e.g., using P2SH for multisignature access control). Using this heuristic, we can uniquely identify a change address in 122 million out of 489 million transactions. Of these change addresses we identified, over 72 million were cases in which the anonymity of *non-multisig* users was reduced because they transacted with a party that used a script-hash address. Over 49 million were cases of script-hash users having their anonymity reduced (i.e., the reverse scenario, in which a payment is made to a regular address).

We note that adding Schnorr signatures [14] could improve confidentiality and privacy of multisignature transactions (e.g., by making multisig indistinguishable from regular addresses) [15]. However, without widespread or mandatory adoption, they could also hurt privacy if they allow further distinction between different users (similar to how distinguishing between the use of script-hash and non-script-hash addresses reduces privacy).

Security. A surprising, but relatively common motif is for multisig users to switch their money from a multisig address to a regular address, and then back into a multisig address. We conjecture that this may happen when users are changing the access control policy on their wallet, although it is unclear why they transfer their funds to a regular address in the interim, and not directly to the new multisig address.

This practice negates some of the security benefits of multisignatures, as it advertises to an attacker when a high-value wallet is most vulnerable. To identify this pattern, we looked for transactions in which all of the inputs were from multisig addresses of the same access structure and there was a single non-multisig output, which was subsequently sent back to a multisig address. We restricted our analysis to single output transactions as this is an indicator of self-churn, i.e. a user

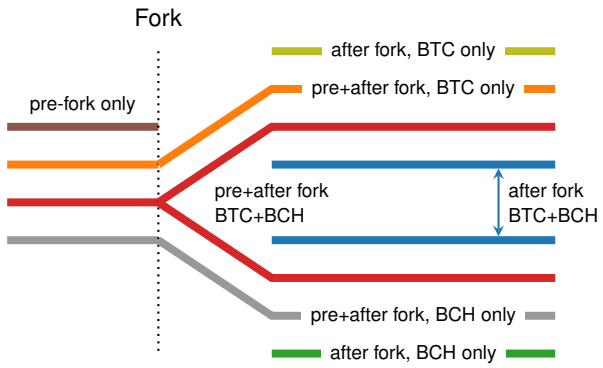

Figure 6: A BTC/BCH address might be used only before the fork (brown), continue to be used only in BTC after the fork (orange), be used only on BCH after the fork (green), etc.

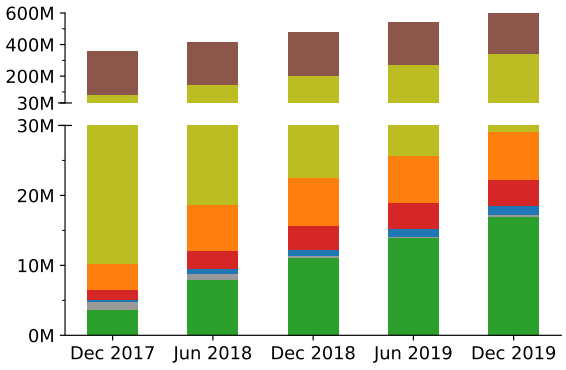

Figure 7: The absolute number of addresses per category. For legend and color coding see Figure 6.

shuffling money among her own addresses.

In Figure 5, we see that a few thousand transactions each month exhibit this pattern, temporarily reducing the security of bitcoins worth up to hundreds of million USD.

3.2 Cashing out on forks hurts privacy

In a blockchain fork, two separate chains emerge with a shared history, often with different rulesets. One prominent example of such a fork is Bitcoin Cash (BCH), which split from the original Bitcoin (BTC) chain in August 2017 over disagreement about the maximum size of blocks (cf. [16]). Users who held BTC at the time of the fork automatically own the same quantity of BCH, too. Unfortunately for users, blockchain forks can also lead to unintentional privacy compromise (cf. [17]). A generally privacy-conscious user who carefully crafts transactions on one chain may perform privacy-harming transactions on another, such as sweeping and cashing out their coins. Here, we investigate privacy implications of the BCH fork.

Preliminaries. We start by systematizing the use of addresses across forked chains (Figure 6). Addresses that held coins before the fork may continue to be used on either (orange or gray) or both chains (red). 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). Addresses may also cease to see use after the fork (brown).

In Figure 7 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.

Privacy impact. We quantify the privacy impact of this activity on BCH using BlockSci's cross-chain clustering capabilities (Figure 8). Cross-chain clustering produces an enhanced clustering of a single chain using additional links from a forked chain, allowing us to investigate the *additional* privacy impact of the behavior on the forked chain. To evaluate the impact of the fork event, 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. Every merge combines two existing (single- or cross-chain) clusters into a cross-chain cluster. 75.44 % of those early merges on BCH occur on the BTC chain (on average, about 8.9 months after occuring 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. With the rough heuristic that each cross-chain cluster represents a distinct user, 99,500 users are affected by this privacy leak: that is, it becomes possible to link their BTC addresses with each other based on their BCH activity.

Next, we evaluate the long-term privacy impact of the fork. To this end, we create a cross-chain clustering of BTC and BCH until Dec 31, 2019. Again, using BCH data to enhance the BTC clustering, we observe a total of 571,924 additional cluster merges from cross-chain clustering. The enhanced clustering includes almost 200,000 cross-chain clusters that contain over 750,000 single-chain clusters of the BTC chain (as some cross-chain clusters may contain multiple single-chain clusters). The cross-chain clusters together contain almost 30 million addresses, or roughly 5 % of all BTC addresses. In other words, roughly 5 % of BTC addresses potentially suffer a privacy compromise due to cash-out behavior on BCH.

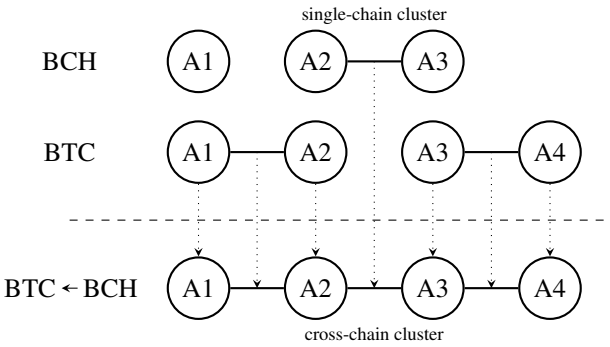

Figure 8: Two single-chain clusters on the BTC blockchain are merged into a cross-chain cluster based on the link between A2 and A3 found in a BCH cluster.

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 6). As of December 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. [18]).

3.3 (In)effectiveness of Bitcoin Core's fee estimation

The Bitcoin protocol defines an upper limit on the size of each block, effectively limiting the number of transactions that miners can include in a block. Bitcoin users compete with each other for the inclusion of their transaction in a block by paying a transaction fee, as rational miners prioritize transactions that pay higher fees per size unit. Demand for block space (equally, the number of pending transactions) is constantly changing, and with it the minimum fee required for transactions to get included in the next block. Supply for block space is not created on a fixed schedule: the arrival time of blocks follows an exponential distribution introduced by the proof-of-work mining mechanism. And the fee mechanism used by Bitcoin, effectively a pay-as-bid auction, is not incentive compatible, making it difficult to determine the optimal fee to pay [19, 20].

Wallets often use fee estimation techniques that use historic data to estimate fees such that transactions get included in the blockchain within a certain target time frame with high confidence. In particular, Bitcoin Core has a fee estimation feature that is well known, widely used, and relied upon. Given a target interval, say 2 blocks, the algorithm chooses fees such that in the past 60 % of transactions were included in half of a target interval, 85 % within the target and at least 95 % within twice the target. But how good is this estimate? Note that most users' time preferences are in terms of time and not blocks. Suppose a user values their transaction being included in a block in 20 minutes or less, and hence selects a transaction fee corresponding to a target of 2 blocks, based on the mean block interval of 10 minutes. How long can the user actually expect to wait? We use BlockSci to answer this question.

Data collection. We used BlockSci's mempool recorder to record timestamps of transactions submitted to the Bitcoin P2P network over a time span of 3 months, from 8/8/19 until 11/19/19. In total, we collected timestamps for 32.98 million transactions, 99.94% of all transactions that were included in the blockchain during that epoch. While the timestamps observed by different nodes may very, this delay is small: compared to timestamps collected by blockchain.com, our timestamps lag by an average of 0.9 ($\pm$ 0.3) seconds.

Bitcoin Core produces fee estimates in two modes: a conservative mode that is supposed to be less impacted by short-term drops in fee levels [21] (the default mode of the RPC interface), and a more aggressive economic mode that is used for transactions using replace-by-fee (RBF), a transaction replacement option allowing users to increase transaction fees after submitting a transaction to the network (the default mode of the wallet GUI). We collected fee estimates in the conservative mode every ten seconds during this time frame. While this mode may err on the side of higher fees (and thus faster inclusion), we chose it because it does not require accounting for replaced transactions, something BlockSci does not track.

Identifying Bitcoin Core transactions. To identify Bitcoin Core transactions, we first filter for transactions that set a non-zero locktime, a characteristic of the Bitcoin Core client [22]. Then, we select transactions that have RBF disabled. This yields a subset of 4,589,246 transactions out of the 32.98 million transactions we collected timestamps for (13.9 %). Next, we identify transactions where the fees paid by the transaction matches the estimate we recorded for one of the common target times. A manual inspection shows very little variance around our recorded estimate, hence we choose to consider all transactions that are within a threshold of $\pm$5 satoshi. If paid fees overlap with estimates for multiple targets, we select the shortest target time. This selection yields 981,214 transactions.

Analyzing calibration. For these transactions we calcu-

late the difference between Bitcoin Core's targeted inclusion times at the 60 %, 85 % and 95 % quantiles and the actual inclusion times (shown in Table 6 in the appendix). Many transactions get included much earlier than targeted (e.g., 60 % of transactions targeting a 60 minute inclusion are included in under 16.75 minutes). However, the 95 % quantile considerably lags behind (e.g., 59 minutes behind twice the target time for transactions with a 60 minute target).

Analyzing variability. However, calibration is not the whole story. We use a regression analysis to better understand how well targeted inclusion time corresponds to actual inclusion times. As the inclusion time is influenced by the block arrival rate, which follows an exponential distribution, we use a Generalized Linear Model with a Gamma distribution (detailed results are provided in Table 10 in the appendix). We include weekly fixed effects to account for gradually changing factors like the hash rate.

Targeted inclusion time explains a mere 17.2 % of the deviance of the model (a measure of fit compared to a perfect model). This means that despite the use of fee estimation, there is a high degree of uncertainty in the actual inclusion time, primarily due to the inherent randomness introduced by the proof-of-work mining but possibly also the unpredictable behavior of other users.

Bitcoin Core incorporates the state of the mempool in a relatively straightforward way for fee estimation: if a transaction resides in the mempool longer than the targeted inclusion interval, its fee is considered as insufficient. But another way to use mempool state is as an estimate of the backlog of transactions. We perform another regression where we incorporate the size of the mempool as a predictor, which gives a rough indication of how much fee estimation might be improved by incorporating mempool information in a more sophisticated way. We see that the deviance explained rises to only 22.4 %, suggesting that the limitation is intrinsic.

We offer two main takeaways from this analysis: Bitcoin users should be careful not to over-rely on the waiting time estimates produced by wallet software, and cryptocurrency researchers and designers should consider alternatives to the pay-as-bid auction employed by Bitcoin that may achieve a tighter relationship between fees and inclusion time.

3.4 Improved estimates of the velocity of cryptocurrencies

The velocity of money is the frequency with which one unit of currency is used for purchases in a unit of time. It can provide an insight into the extent to which money is used as a medium of exchange versus a store of value.

In most cases it is not possible to infer the purpose behind a cryptocurrency transaction from the blockchain. However, an alternative definition of the velocity of money is the frequency with which one unit of currency changes possession in any manner (whether or not for purchases of goods and services) in a unit of time. Blockchain analysis may enable estimating the velocity of cryptocurrencies under this definition.

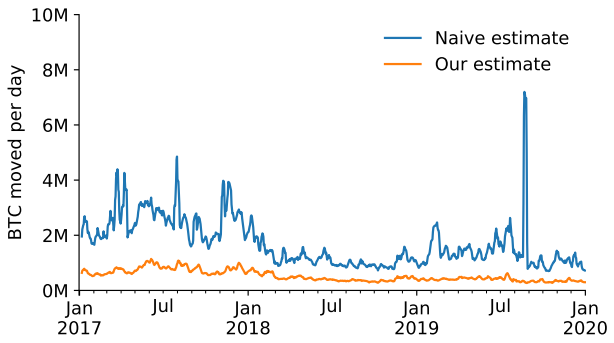

Figure 9: Two estimates of the velocity of bitcoins.

Even under this simplified definition, it is challenging to estimate the velocity of cryptocurrencies. A naive method would be to compute the total value of transaction outputs in a unit of time and divide it by the total value of the money supply during that period. However, multiple addresses may be controlled by the same entity, and therefore not all transaction outputs represent changes in possession. Meiklejohn et al. call this "self-churn" [3], a term that we adopt. The impact of self-churn is visually obvious in the graph of total transaction outputs (Figure 9). We would not expect spikes such as those in early 2017 if the graph reflected actual money demand, which would be much more stable over time.

To minimize the effect of self-churn, we adopt two heuristics. First, we eliminate outputs controlled by an address that can be linked to one of the inputs' addresses (through address clustering, cf. Section 2.4), ignoring "superclusters" to minimize false positives. This reduces change outputs and transactions that are detectable as an entity "shuffling their money around". We also eliminate outputs that are spent within less than k blocks (we use $k = 4$). Manual examination suggests that such transactions are highly likely to represent self-churn, such as "peeling chains" where a large output is broken down into a series of smaller outputs in a sequence of transactions.

The orange line in Figure 9 shows the daily transaction volume on the Bitcoin blockchain after applying the above two heuristics. With this estimate, the velocity of Bitcoin works out to 1.2 per month averaged over the period January 2017–June 2018, compared to 3.9 with the naive metric, and 0.7 over the period July 2018–December 2019, compared to 2.2 with the naive metric. Our revised estimate is not only much lower but also much more stable over time.

Starting in 2018 the naive estimate drops closer to our improved estimate. We suppose that this is partially due to scarcity in block space (and a corresponding rise in transaction fees), which encourages intermediaries to batch multiple payments into a single transaction, thereby eliminating some

of the self-churn that is evident in the naive estimate earlier. Spikes in the graph, like the one in mid 2019, may represent large intermediaries (e.g., exchanges) moving large amounts of bitcoin to addresses with updated access control structures.

We note several caveats. First, this still likely fails to exclude some transfers of value between addresses controlled by the same entity. Without ground truth, it is hard to be certain how good the estimate is. Second, it doesn't count transfers of possession that don't touch the blockchain. When exchanges, online wallets, and other intermediaries hold money on behalf of users, payments and transfers of "bitcoins" might happen even though no actual bitcoins changed hands (as only account balances in an internal database need to be updated). Nevertheless, we believe that the metric can be a useful proxy for understanding the use of cryptocurrencies, and possibly for comparing between cryptocurrencies.

3.5 Other applications of BlockSci

Besides our own use, BlockSci has seen a variety of use in both academic and industry settings. We are currently aware of at least 9 peer-reviewed articles, 6 preprints, and 2 software projects that use BlockSci for blockchain analysis (a full list is available online[6]).

The dual topics of privacy and forensics are common among these papers. These include information leaks from payments and purchases through intermediaries [4], the use of intermediaries to convert between cryptocurrencies [12], as well as the identification of entities and the analysis of their behavior in the transaction graph [6, 11, 23–25]. Many of these results are of interest to law enforcement and regulators, and we have helped regulators use BlockSci for their own investigations. Two other themes are issues surrounding the security and scalability of cryptocurrencies [26–28], as well as economic analyses of cryptocurrencies [29].

BlockSci has also been used as the foundation for specialized blockchain analysis tools. Boshmaf, Al Jawaheri, and Al Sabah [23] have built a tagging system on top of BlockSci, and the GraphSense blockchain analytics platform uses BlockSci's parser and altcoin support to generate an address graph out of the transaction graph [30].

4 Conclusion

There is a high level of interest in blockchain analysis among developers, researchers, and students, leading to an unmet need for effective analysis tools. While general-purpose in-memory graph databases exist, a tool customized to blockchain data can take advantage of its append-only nature as well as provide integrated high-performance routines for common tasks such as address linking.

[6] https://citp.github.io/BlockSci/studies/

BlockSci has already been widely used as a research and educational tool. We hope it will continue to be broadly useful, and plan to keep maintaining it as open-source software.

Acknowledgments

We are grateful to Lucas Mayer for prototype code, Danny Yuxing Huang, Pranay Anchuri, Shaanan Cohney, Rainer Böhme, Michael Fröwis, Jakob Hollenstein, Jason Anastasopoulos, Sarah Meiklejohn, and Dillon Reisman for useful discussions, and Chainalysis for providing access to their Reactor tool. We also thank the anonymous USENIX Security reviewers, the reviewers of the artifact evaluation process and our shepherd Anita Nikolich for their feedback.

This work is supported by NSF grants CNS-1421689 and CNS-1651938, a grant from the Ripple University Blockchain Research Initiative, the European Union's Horizon 2020 research and innovation programme under grant agreement No. 740558, the Austrian FFG's KIRAS programme under project VIRTCRIME, and an NSF Graduate Research Fellowship under grant number DGE-1148900.

References

[1] Frank McSherry, Michael Isard, and Derek Gordon Murray. "Scalability! But at what COST?" In: *Proceedings of the 15th Workshop on Hot Topics in Operating Systems (HotOS XV)*. May 2015. URL: https://www.usenix.org/conference/hotos15/workshop-program/presentation/mcsherry (visited on 06/12/2020).

[2] Facebook Database Engineering Team. *RocksDB. A persistent key-value store for fast storage environments*. Version 6.10.2. June 5, 2020. URL: https://rocksdb.org/.

[3] Sarah Meiklejohn et al. "A Fistful of Bitcoins: Characterizing Payments Among Men with No Names". In: *Proceedings of the 2013 Internet Measurement Conference (IMC)*. Oct. 2013. DOI: 10.1145/2504730.2504747.

[4] Steven Goldfeder et al. "When the cookie meets the blockchain: Privacy risks of web payments via cryptocurrencies". In: *Proceedings on Privacy Enhancing Technologies*. Vol. 2018. Oct. 2018. DOI: 10.1515/popets-2018-0038.

[5] Wenzel Jakob. *Lock-free parallel disjoint set data structure*. June 14, 2020. URL: https://github.com/wjakob/dset.

[6] Michael Fröwis et al. "Safeguarding the Evidential Value of Forensic Cryptocurrency Investigations". In: (July 28, 2019). arXiv: 1906.12221.

[7] Wenzel Jakob. *pybind11 — Seamless operability between C++11 and Python*. Version 2.5.0. Mar. 31, 2020. URL: https://github.com/pybind/pybind11.

[8] Martin Fowler. *FluentInterface*. Dec. 20, 2020. URL: https://www.martinfowler.com/bliki/FluentInterface.html (visited on 02/14/2020).

[9] Harry Kalodner et al. *BlockSci: Design and applications of a blockchain analysis platform*. Sept. 8, 2017. arXiv: 1709.02489.

[10] Alex Biryukov and Daniel Feher. "Privacy and Linkability of Mining in Zcash". In: *2019 IEEE Conference on Communications and Network Security (CNS)*. June 2019. DOI: 10.1109/CNS.2019.8802711.

[11] Danny Yuxing Huang et al. "Tracking Ransomware End-to-end". In: *Proceedings of the 39th IEEE Symposium on Security & Privacy (S&P)*. May 2018. DOI: 10.1109/SP.2018.00047.

[12] Haaroon Yousaf, George Kappos, and Sarah Meiklejohn. "Tracing Transactions Across Cryptocurrency Ledgers". In: *Proceedings of the 28th USENIX Security Symposium (USENIX Security)*. Aug. 2019. URL: https://www.usenix.org/conference/usenixsecurity19/presentation/yousaf (visited on 06/13/2020).

[13] Rosario Gennaro, Steven Goldfeder, and Arvind Narayanan. "Threshold-Optimal DSA/ECDSA Signatures and an Application to Bitcoin Wallet Security". In: *Proceedings of the 14th International Conference on Applied Cryptography and Network Security (ACNS)*. Vol. 9696. Lecture Notes in Computer Science (LNCS). June 2016. DOI: 10.1007/978-3-319-39555-5_9.

[14] Claus-Peter Schnorr. "Efficient signature generation by smart cards". In: *Journal of Cryptology* 4 (1991). DOI: 10.1007/BF00196725.

[15] Bitcoin Core. *Technology roadmap - Schnorr signatures and signature aggregation*. URL: https://bitcoincore.org/en/2017/03/23/schnorr-signature-aggregation/ (visited on 06/07/2020).

[16] Yujin Kwon et al. "Bitcoin vs. Bitcoin Cash: Coexistence or Downfall of Bitcoin Cash?" In: *Proceedings of the 40th IEEE Symposium on Security & Privacy (S&P)*. May 2019. DOI: 10.1109/SP.2019.00075.

[17] Abraham Hinteregger and Bernhard Haslhofer. "Short Paper: An Empirical Analysis of Monero Cross-chain Traceability". In: *Proceedings of the 23th International Conference on Financial Cryptography and Data Security (FC)*. Vol. 11598. Lecture Notes in Computer Science (LNCS). Feb. 2019. DOI: 10.1007/978-3-030-32101-7_10.

[18] Francisco Memoria. *Bitcoin Gold Wallet Scam Sees Fraudsters Steal $3.2 Million*. CCN Markets. Nov. 24, 2017. URL: https://www.ccn.com/bitcoin-gold-wallet-scam-nets-fraudsters-3-2-million-after-stealing-users-private-keys/ (visited on 02/14/2020).

[19] Ron Lavi, Or Sattath, and Aviv Zohar. "Redesigning Bitcoin's fee market". In: *Companion Proceedings of the The Web Conference (WWW) 2019*. May 2019. DOI: 10.1145/3308558.3313454.

[20] Soumya Basu et al. *Towards a Functional Fee Market for Cryptocurrencies*. DOI: 10.2139/ssrn.3318327.

[21] Bitcoin Core. *estimatesmartfee (0.19.0 RPC)*. Version 0.19.0. URL: https://bitcoincore.org/en/doc/0.19.0/rpc/util/estimatesmartfee/ (visited on 02/15/2020).

[22] Peter Todd. *Discourage fee sniping with nLockTime. Pull Request #2340*. Dec. 19, 2014. URL: https://github.com/bitcoin/bitcoin/pull/2340 (visited on 06/14/2020).

[23] Yazan Boshmaf, Husam Al Jawaheri, and Mashael Al Sabah. "BlockTag: Design and Applications of a Tagging System for Blockchain Analysis". In: *Proceedings of the 34th IFIP TC11 Information Security Conference & Privacy Conference*. June 2019. DOI: 10.1007/978-3-030-22312-0_21.

[24] Marc Jourdan et al. "Characterizing Entities in the Bitcoin Blockchain". In: *2018 IEEE International Conference on Data Mining Workshops (ICDMW)*. Oct. 2018. DOI: 10.1109/ICDMW.2018.00016.

[25] Yury Zhauniarovich et al. *Characterizing Bitcoin donations to open source software on GitHub*. July 9, 2019. arXiv: 1907.04002.

[26] Iain Stewart et al. "Committing to quantum resistance: a slow defence for Bitcoin against a fast quantum computing attack". In: *Royal Society Open Science* 5.6 (6 June 2018). DOI: 10.1098/rsos.180410.

[27] Cristina Pérez-Solà et al. "Another coin bites the dust: an analysis of dust in UTXO-based cryptocurrencies". In: *Royal Society Open Science* 6.1 (1 Jan. 2019). DOI: 10.1098/rsos.180817.

[28] Cristina Pérez-Solà et al. *Analysis of the SegWit adoption in Bitcoin*. URL: https://deic-web.uab.cat/~guille/publications/papers/2018.recsi.segwit.pdf (visited on 06/13/2020).

[29] Bruno Biais et al. *Equilibrium Bitcoin Pricing*. DOI: 10.2139/ssrn.3261063.

[30] Bernhard Haslhofer, Roman Karl, and Erwin Filtz. "O Bitcoin Where Art Thou? Insight into Large-Scale Transaction Graphs". In: *Joint Proceedings of the Posters and Demos Track of the 12th International Conference on Semantic Systems and the 1st International Workshop on Semantic Change & Evolving Semantics (SuCCESS'16)*. Sept. 13, 2016. URL: http://ceur-ws.org/Vol-1695/paper20.pdf (visited on 06/14/2020).

[31] Neo4j Labs. *Awesome Procedures On Cypher (APOC)*. May 23, 2020. URL: https://neo4j.com/labs/apoc/ (visited on 06/14/2020).

Table 6: Bitcoin Core fee estimates are chosen such that 60 % of past transactions were included within half of the target interval, 85 % included within the target interval, and 95 % within twice the target interval. The differences of actual inclusion times with those targeted inclusion times of transactions are shown below.

Target T	Difference to target (in minutes)		
	60 % ($0.5T$)	85 % (T)	95 % ($2T$)
20 min	+0.48	+3.63	+5.52
40 min	-7.00	-2.90	+18.47
60 min	-13.25	-2.52	+58.92
120 min	-38.25	-26.67	+108.64
240 min	-96.53	-126.35	-41.58

A Graph Database Comparison

We model a simplified transaction graph that contains all important types of nodes but does not include many of the properties that BlockSci provides access to (the resulting graph thus requires less storage than a full layout would require). While many different graph layouts are conceivable, we chose a layout that models the transaction graph as explicitly as possible (cf. Figure 10). There are four types of nodes: blocks, transactions, outputs and addresses. Blocks reference included transactions, transactions reference previous outputs that are being spent as well as new outputs being created, and outputs reference the address they send value to.

Table 7 shows the Cypher queries that we used. Minor syntax changes were necessary due to the particularities of the respective import scripts.

Table 8 contains the full measurements of our performance analysis. All benchmarks are run on an *r5.8xlarge* EC2 instance (32 vCPUs, 256 GiB memory, 800 GiB EBS volume).

Table 7: Cypher queries used in the graph database performance comparison

Tx locktime > 0
```
MATCH (tx:Tx) WHERE tx.locktime > 0 RETURN COUNT(*)
```
Max output value
```
MATCH (o:Output) RETURN MAX(o.value)
```
Calculate fee
```
MATCH (i)<-[:TX_INPUT]-(tx:Tx)
WITH tx, SUM(i.value) as totalIn
MATCH (tx)-[:TX_OUTPUT]->(o)
WITH tx, (totalIn - SUM(o.value)) as fee
RETURN MAX(fee)
```
Satoshi Dice address
```
MATCH (a)<-[:TO_ADDRESS]-(o) WHERE ID(a) = {}
RETURN SUM(o.value)
```
Zero-conf outputs
```
MATCH (b:Block)-[:CONTAINS]->()-[:TX_OUTPUT]->(o)
<-[:TX_INPUT]-()<-[:CONTAINS]-(b)
RETURN COUNT(o)
```
Zero-conf outputs *(Memgraph)*
```
MATCH (b1:Block)-[:CONTAINS]->()-[:TX_OUTPUT]->(o)
<-[:TX_INPUT]-()<-[:CONTAINS]-(b2)
WHERE b1 = b2 RETURN COUNT(o)
```
Locktime change
```
MATCH (tx:Tx)-[:TX_OUTPUT]->(o)<-[:TX_INPUT]-(tx2)
WHERE (tx.locktime > 0) = (tx2.locktime > 0)
WITH tx, COUNT(o) as cnt WHERE cnt = 1
RETURN COUNT(*)
```
Locktime change *(RedisGraph)*
```
MATCH (tx:Tx)-[:TX_OUTPUT]->(o)<-[:TX_INPUT]-(tx2)
WHERE (tx.locktime = 0 AND tx2.locktime = 0)
OR (tx.locktime > 0 AND tx2.locktime > 0)
WITH tx, COUNT(o) as cnt WHERE cnt = 1
RETURN COUNT(*)
```

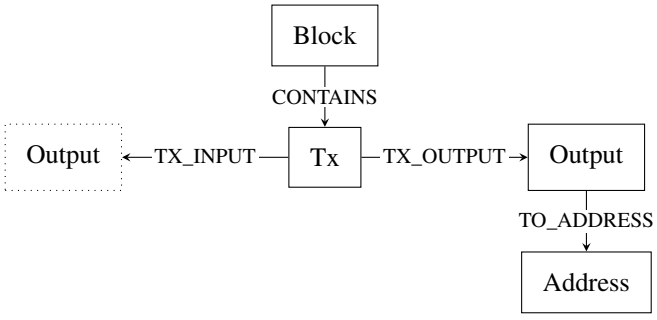

Figure 10: Property graph model

Table 8: Average running time in seconds and standard deviation (in parentheses) over five consecutive runs for various graph queries and data set sizes on a *r5.8xlarge* EC2 instance (32 vCPUs, 256 GiB memory). We used Neo4j v3.5.14, RedisGraph v2.0.1 (running on top of Redis v5.0.7) and Memgraph v0.15.0. Fluent interface is single-threaded.

Query	BlockSci			Neo4j		RedisGraph	Memgraph
	C++ (ST)	C++ (MT)	Fluent interface	w/o index	w/ index		
12.5M transactions							
Tx locktime > 0	0.15 (0.0)	0.01 (0.0)	0.72 (0.0)	3.78 (0.8)	0.01 (0.0)	0.93 (0.0)	5.35 (0.1)
Max output value	0.23 (0.0)	0.02 (0.0)	1.96 (0.0)	13.46 (0.7)	14.28 (0.1)	2.21 (0.0)	19.52 (0.4)
Calculate fee	0.29 (0.0)	0.02 (0.0)	1.51 (0.0)	131.21 (2.4)	132.95 (1.3)	–[1]	81.87 (2.2)
Satoshi Dice address	0.22 (0.0)	–[3]	0.24 (0.0)	0.46 (0.0)	0.46 (0.0)	1.06 (0.0)	21.17 (0.1)
Zero-conf outputs	2.58 (0.0)	0.16 (0.0)	8.48 (0.1)	92.35 (0.3)	93.77 (0.1)	601.21 (0.3)	32.47 (NA)
Locktime change	3.49 (0.0)	0.20 (0.0)	8.55 (0.0)	96.61 (0.9)	100.29 (0.9)	–[1]	47.33 (1.8)
25M transactions							
Tx locktime > 0	0.31 (0.0)	0.03 (0.0)	1.37 (0.0)	7.84 (1.4)	0.05 (0.1)	1.85 (0.0)	16.44 (0.2)
Max output value	0.46 (0.0)	0.03 (0.0)	3.91 (0.0)	26.63 (0.0)	24.55 (2.9)	4.48 (0.0)	40.08 (0.5)
Calculate fee	0.57 (0.0)	0.03 (0.0)	2.79 (0.1)	302.73 (6.7)	303.69 (6.3)	–[1]	187.02 (4.9)
Satoshi Dice address	0.49 (0.0)	–[3]	0.54 (0.0)	0.95 (0.0)	0.99 (0.0)	2.56 (0.1)	45.91 (0.4)
Zero-conf outputs	5.47 (0.0)	0.32 (0.0)	18.17 (0.3)	192.01 (0.9)	207.41 (1.7)	1488.94 (2.7)	59.96 (0.3)
Locktime change	7.57 (0.0)	0.45 (0.0)	18.21 (0.0)	208.95 (0.9)	213.59 (1.8)	–[1]	122.98 (3.6)
50M transactions							
Tx locktime > 0	0.68 (0.0)	0.05 (0.0)	2.90 (0.1)	15.86 (2.3)	0.05 (0.1)	3.69 (0.0)	–[2]
Max output value	0.98 (0.0)	0.05 (0.0)	8.79 (0.1)	63.77 (1.3)	61.92 (5.5)	10.08 (0.1)	–[2]
Calculate fee	1.13 (0.0)	0.06 (0.0)	5.20 (0.0)	–[1]	–[1]	–[1]	–[2]
Satoshi Dice address	0.55 (0.0)	–[3]	0.60 (0.0)	1.05 (0.0)	1.08 (0.0)	7.34 (0.3)	–[2]
Zero-conf outputs	13.01 (0.0)	0.78 (0.0)	41.02 (0.5)	472.20 (1.2)	493.25 (1.9)	5716.33 (8.8)	–[2]
Locktime change	18.68 (0.0)	1.11 (0.0)	42.17 (0.1)	551.40 (4.1)	558.81 (4.5)	–[1]	–[2]
100M transactions							
Tx locktime > 0	1.44 (0.0)	0.09 (0.0)	5.57 (0.1)	–	–	–	–
Max output value	2.02 (0.0)	0.11 (0.0)	19.07 (0.1)	–	–	–	–
Calculate fee	2.30 (0.0)	0.12 (0.0)	10.55 (0.0)	–	–	–	–
Satoshi Dice address	0.54 (0.0)	–[3]	0.60 (0.0)	–	–	–	–
Zero-conf outputs	29.36 (0.0)	1.71 (0.0)	92.47 (0.9)	–	–	–	–
Locktime change	42.65 (0.0)	2.53 (0.0)	90.10 (0.2)	–	–	–	–
200M transactions							
Tx locktime > 0	2.71 (0.0)	0.18 (0.0)	11.60 (0.9)	–	–	–	–
Max output value	3.92 (0.0)	0.21 (0.0)	35.91 (0.7)	–	–	–	–
Calculate fee	4.50 (0.0)	0.23 (0.0)	19.26 (0.1)	–	–	–	–
Satoshi Dice address	0.55 (0.0)	–[3]	0.60 (0.0)	–	–	–	–
Zero-conf outputs	60.48 (0.0)	4.56 (0.0)	175.37 (1.4)	–	–	–	–
Locktime change	98.22 (0.1)	6.62 (0.1)	181.23 (0.6)	–	–	–	–

–: not measured, –[1]: did not finish in reasonable time (based on other queries and dataset sizes), –[2]: ran out of memory, –[3]: not applicable
w/ index: property indexes created for `Tx.locktime` and `Output.value`
ST = single-threaded, MT = multithreaded

Table 9: Database sizes on disk and when loaded in memory during the benchmark, in GB. Memory consumption is measured after data has been loaded but before queries have been executed. Additional memory may be required to run the queries. For BlockSci, memory usage is lower than storage on disk as not all data is loaded into memory. For Neo4j, the whole graph was loaded into memory using the APOC warmup script [31] before executing queries for optimal performance.

		BlockSci *		Neo4j		RedisGraph		Memgraph	
Txs	Block height	Disk	Memory	Disk	Memory	Disk	Memory	Disk	Memory
12.5 M	220 406	3.5	1.3	6	7.1	3.5	20	4.7	56
25 M	262 176	7.2	2.6	12	13.4	7	41	9.6	114
50 M	327 439	17.5	5.7	27	28.5	16	97	–	–
100 M	390 069	38.4	12.1	58	60.2	–	–	–	–
200 M	454 860	80.9	23.2	110	113.6	–	–	–	–

*Denotes the size of the full BlockSci Data (excluding parser state).
The other databases use a simplified data model.

Table 10: GLM regression of the time until inclusion for transactions (in minutes) with and without the current size of the mempool and weekly fixed effects, fitted using a Gamma distribution with identity link function.

	target only	target + FE	w/ mempool + FE
(Intercept)	4.1626 ***	-1.2849 ***	-5.3453 ***
	(0.098)	(0.390)	(0.318)
Target time	0.54741 ***	0.4955 ***	0.2981 ***
	(0.003)	(0.003)	(0.002)
Mempool size			0.0019 ***
			(<0.001)
Weekly fixed effects	No	Yes	Yes
Deviance explained	0.1591	0.1724	0.2238
Nagelkerke R^2	0.2002	0.2163	0.2774
N	981 212	981 212	981 212

(. . .) = standard error. Significance level code: ***p<0.001.